CONCURRENT PROGRAMS ARE EASIER TO VERIFY THAN SEQUENTIAL PROGRAMS

Edmund M. Clarke*

Department of Computer Science
Duke University

July 20, 1978

CS-1978-6

ABSTRACT

We investigate the problem of automatically synthesizing correctness proofs for synchronization skeletons of concurrent programs. These synchronization skeletons are expressed in a simple concurrent programming language (SCL) in which logically related variables accessed by more than one process are grouped together as resources. Correctness proofs for SCL programs are expressed in a proof system similiar to the system of Hoare (HO72) and Owicki and Gries (OW76). Proofs of synchronization properties are constructed by devising predicates called resource invariants which describe relationships among the variables of a resource when no process is in a critical region for the resource. In the system of Hoare-Owicki-Gries, these resource invariants must be supplied by the programmer. We show that for a large class of SCL programs it is possible to automatically generate appropriate resource invariants directly from the text of the program.

* Supported by National Science Foundation Grant No. MCS-7508146.

CONCURRENT PROGRAMS ARE EASIER TO VERIFY THAN SEQUENTIAL PROGRAMS

1.1 Background. Researchers in program verification have often argued that it is premature to develop methods for proving the correctness of concurrent programs. The argument usually runs as follows: "Since sequential programs are a subset of concurrent programs, how can we expect to develop methods for verifying concurrent programs when practical methods do not yet exist for the verification of sequential programs." If we view sequential programs as a subclass of concurrent programs, and if we attempt to prove the same type of correctness results for concurrent programs as for sequential programs (e.g. program P computes function F), then the above argument may be correct. On the other hand, if we abstract from a concurrent program that portion of the program which deals with synchronization (the synchronization skeleton), and if we restrict correctness proofs to synchronization properties (e.g. absence of deadlock), then it is possible to argue that concurrent programs are easier to verify than sequential programs. Three reasons in support of this view are:

(1) The synchronization skeleton of a concurrent program is generally very small in comparison to the entire program. A crude count of the UNIX operating system indicates that less then 5% of the total statements are synchronization statements. Because long concurrent programs need not have long synchronization skeletons, we believe that the arguments expressed in (DE77) regarding the feasibility of correctness proofs for sequential programs are not applicable to proofs of correctness for synchronization skeletons.

(2) In many cases, it is relatively easy to isolate the synchronization skeleton of a concurrent program from the sequential part of the program. Consider, for example, the typical producer-consumer program in which two processes communitcate via a message buffer; here the synchronization skeleton is independent of the sequential producer and consumer processes.

(3) A large class of synchronization problems are counting problems. Thus, the verification conditions generated in the correctness proofs can be expressed in terms of linear inequalities involving the synchronization variables, and techniques such as integer programming can be exploited to determine their validity.

What verification techniques are best suited for proving the correctness of synchronization skeletons? Hoare (HO72), and Owicki and Gries (OW76) have developed a proof system for conditional critical regions in which logically related variables which must be accessed by more than one process are grouped together as resources. Individual processes are allowed to access a resource only within a conditional critical region for that resource. Proofs of synchronization properties are constructed by devising predicates called resource invariants which describe the relationship among the variables of a resource when no process is within a critical section for the resource. Related methods for verifying concurrent programs which use resource invariants have been described by Habermann (HA72), Lauer (LU72), Keller (KE76), Lamport (LM77), and Pneuli (PN77).

In constructing correctness proofs for synchronization skeletons using the proof system of Hoare-Owicki-Gries, the programmer is required to supply the resource invariants. In this paper we investigate the problem of automatically synthesizing resource invariants for synchronization skeletons of concurrent programs. The synchronization skeletons are expressed in a simple concurrent programming language (SCL) in which parallelism is introduced via "cobegin...coend" blocks and processes access shared data via conditional critical regions. This paper will consider only invariance (PN77) or safety properties (LA77) of SCL programs. This class of properties includes mutual exclusion and absence of deadlock and is analoguous to partial correctness of sequential programs. Correctness proofs of SCL programs are expressed in a

proof system similar to the system of Hoare-Owicki-Gries; we will refer to such proofs as resource invariant proofs.

1.2 New results of this paper. To gain insight on synthesis methods for resource invariant proofs we restrict the SCL language so that all processes are nonterminating loops of the form "cycle $S_1,\ldots,S_n$ end" and the only statements allowed in a process are P and V operations on semaphores. We call this class of SCL programs PV programs. For PV programs there is a simple method for generating resource invariants, i.e. the semaphore invariant method of Habermann (HA72) which expresses the current value of a semaphore in terms of its initial value and the number of P and V operations which have been executed. Although the semaphore invariant is simple to state, it is quite powerful as a technique for proving PV programs. We show that the semaphore invariant method is at least as powerful as the reduction method of Lipton (LI75) for proving freedom from deadlock.

The semaphore invariant method, however, is not complete for proving either absence of deadlock or mutual exclusion of PV programs. We prove that it is possible to devise PV programs for which deadlock (mutual exclusion) is impossible, but the semaphore invariant method is not sufficiently powerful to establish this fact. This incompleteness result is important because it demonstrates the role of convexity in the generation of powerful resource invariants. We also give a characterization of the class of PV programs for which the semaphore invariant method is complete for proving absence of deadlock (mutual exclusion).

The semaphore invariant method can be generalized to the class of linear SCL programs in which solutions to many standard synchronization problems including the dining philosopher's problem, the reader's writers problem and the cigarette smoker's problem can be expressed. Although the generalized

semaphore invariant fails to be complete for exactly the same reason as the standard semaphore invariant method, it is sufficiently powerful to permit proofs of mutual exclusion and absence of deadlock for a significant class of concurrent programs (e.g. an implementation of the reader and writers problem with writer priority).

When the generalized semaphore invariant is not sufficiently powerful to prove some desired property of an SCL program, is it possible to synthesize a stronger resource invariant? We argue that resource invariants are fixedpoints, and that by viewing them as fixedpoints it is possible to automatically generate invariants which are considerably stronger than the semaphore invariants previously described. Specifically we show that the resource invariants of an SCL program C are fixedpoints of a functional F_C which can be obtained in a natual manner from the text of program C. The least fixed point $\mu(F_C)$ of F_C is the "strongest" such resource invariant, e. g. if the program C is free of deadlock then this fact may be established using $\mu(F_C)$. Since the functional F_C is continuous, the least fixedpoint $\mu(F_C)$ may be expressed as the limit $\mu(F_C) = \bigvee_{j=1}^{\infty} F_C^j(\text{false})$. Because of the infinite disjunction in the formula for $\mu(F_C)$, this characterization of $\mu(F_C)$ cannot be used directly to compute $\mu(F_C)$ unless C has only a finite number of possible different states.

By using the notion of widening of Cousot (CO77) however, we are able to speed up the convergence of the chain $F_C^j(\text{false})$ and obtain a close approximation to $\mu(F_C)$ in a finite number of steps. The widening operator which we use exploits our earlier observation on the importance of convexity in the generation of resource invariants. Examples are given in the text to illustrate the power of this new technique for generating resource invariants. Although fixedpoint techniques have been previously used in the study of resource invariants ((LA76), (FL77)), we believe that this is the first research on methods for speeding up the convergence of the sequence of approximations to $\mu(F_C)$.

1.3 Outline of paper. Section 2 contains a description of the concurrent programming language SCL which will be used tc illustrate our ideas on the automatic sysnthesis of resource invariants. In Section 3 a proof system is described for SCL programs; this proof system is based on the use of resource invariants and is similar to the system of Hoare-Owicki-Gries. The semaphore invariant method and its generalization to linear SCL programs are discussed in Sections 4, 5, and 6. In Section 7 fixedpoint techniques are described which enable a resource invariant to be expressed as the limit of a sequence of successively better approximations. Section 8 presents the technique for speeding up the convergence of the sequence of approximations to a resource invariant. The paper concludes with a discussion of the results and additional open problems in Section 9.

2. A simple concurrent programming language (SCL). An SCL program will consist of two parts:

(1) an initialization part

v_1:=e; v_2:=e_2;...; v_t:=e_t

in which initial values are assigned to the program variables $v_1,\ldots v_t$, and

(2) a concurrent execution part

resource $R_1(v_1^1,\ldots,v_{m_1}^1),\ldots,R_r(v_1^r,\ldots,v_{m_r}^r)$:

cobegin P1//P2//...Pn end

which permits the simultaneous or interleaved execution of the statements in the processes P1,...Pn.

Logically related sets of variables which must be accessed by more than one process are grouped together (e.g. $R_1(v_1^1,\ldots v_{m_1}^1)$) in the resource prefix of the "cobegin...coend" statement.

Processes have the form

Pi:cycle S_1^i; S_2^i;...; $S_{k_i}^i$ end

where S_1^i; $S_2^i\ldots S_{k_i}^i$ is a list of synchronization statements. The "cycle" con-

struct is a nonterminating loop with the property that the next statement to be executed after $S^i_{k_i}$ is the first statement S^i_1 of the loop. Although the "cycle" statment simplifies the generation of loop invariants, we will also treat terminating loops (e.g. while loops) in a later paper (CL78).

Synchronization statements are conditional critical regions of the form "with R when b do A od" where R is one of the resources listed in the prefix of the "cobegin...coend" statement. Only variables listed in R can appear in the boolean expression b and the body A of the conditional critical region.

When execution of a process reaches the conditional critical region "with R when b do A od" the process is delayed until no other process is using resource R and the condition b is satisfied. When the process has control of R and b is satisfied, the statement A may be executed.

3. Resource invariant proofs. In this Section we adapt the proof system of Hoare-Owicki-Gries to SCL programs. We will use the standard triple notation {P} A {Q} of Hoare (HO69) to express the partial correctness of the sequential statement A with respect to the precondition P and postcondition Q. SP[A](P) will denote the strongest postcondition (CL77) corresponding to statement A and precondition P. Proof systems for partial correctness of sequential statements have been widely discussed in the literature and will not be further discussed in this paper.

Let C be an SCL program and let ST be the set of statements occurring within the processes of C. A resource invariant system RS_C for C will consist of two parts:

(A) A set $IR_1,...IR_r$ of predicates (the resource invariants) corresponding to the r resources in the resource prefix of C.

(B) Proofs of sequential correctness for each of the individual processes of C. For our purposes, these correctness proofs will be represented by a set

(2) If S^i_j is a conditional critical region with associated auxiliary variable c^i_j then $post(S^i_j) = pre(S^i_j)\ [(c^i_j - 1)/c^i_j]$

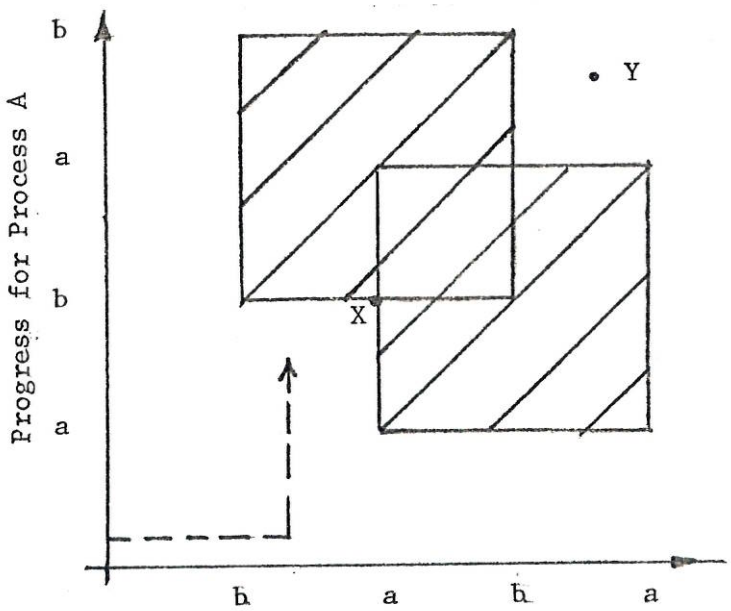
This technique for adding auxiliary variables and generating verification conditions will be used throughout the paper and will be called the canonical annotation associated with the program C.

For a PV program C, we will refer to the resource invariant system consisting of the semaphore invariants for each of the semaphores in the program and the canonical annotation as the semaphore invariant system corresponding to C.

5. Incompleteness of the semaphore invariant method. The incompleteness of the semaphore invariant method is best explained by means of progress graphs (DI67). The progress graph is a graphical method for representing the feasible computations of a PV program. Consider, for example, the program C:

```
a:=1; b:=1
cobegin
   A:cycle P(a); P(b); V(a); V(b) end
   //
   B:=cycle P(b); P(a); V(b); V(a) end
end
```

Feasible computations of this program can be represented by a graph in which the number of instructions executed by a process is used as a measure of the progress of the process, e.g.

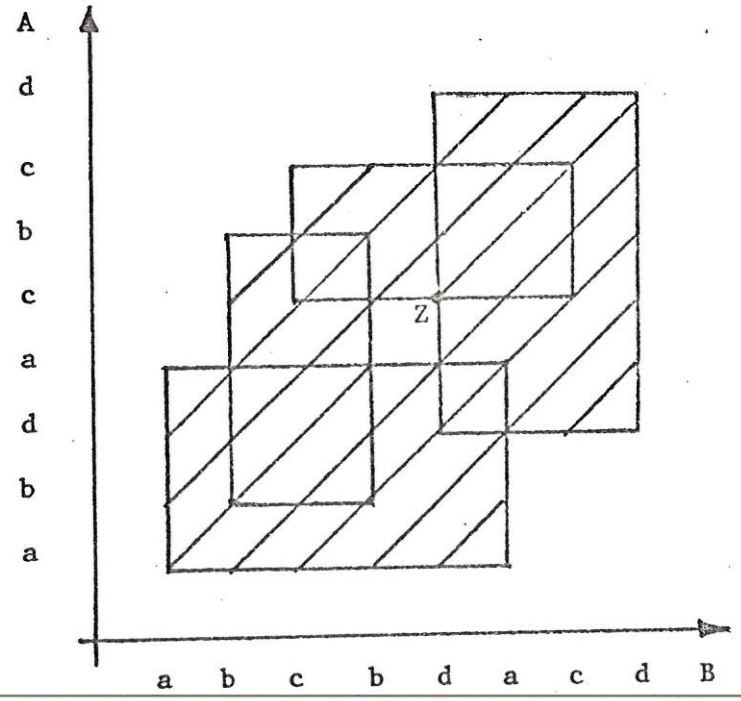


The dashed line represents a computation of the program C in which process B executes P(b) and process A executes P(a). The shaded region of the graph respresents those program states which fail to satisfy the semaphore invariants for a or b; such states are called unfeasible states. The point labeled X in the graph is a deadlock state; further progress for either process A or process B would violate either the invariant for semaphore a or the invariant for semaphore b. Those points in the graph (states of C) which are not reachable from the origin (initial state) by a polygonal path composed of horizontal and vertical line segments which never cross an unfeasible region (by a valid computation sequence of C) are called an unreachable points (states). All unfeasible points are unreachable. The point labeled Y in the graph is an example of an unreachable feasible point; if the program C were started in state Y, the semaphore invariants would not be violated.

Consider the PV program C:

```
a:=b:=c:=d:=1
cobegin
   A:cycle P(a); P(b); P(d): V(a); P(c); V(b); V(c); V(d) end
   //
   B:cycle P(a); P(b); P(c); V(b); P(d); V(a); V(c); V(d) end
coend
```

The progress graph for C is shown below:



Let SI_C be the semaphore invariant system for the program C. It is not difficult to show that the condition $D(SI_C)$ for absence of deadlock given in Section 3 is satisfiable. Thus absence of deadlock cannot be proven by means of the semaphore invariant method. From the progress graph for C we observe that deadlock can never occur during an execution of the program C. The state Z which satisfies $D(SI_C)$ is an example of an unreachable feasible state which is also a deadlock state; we will call such states trap states.

In a later paper (CL78), the terms progress graph, trap states, etc. will be extended to SCL programs with more than two processes and the following characterization of the semaphore invariant method will be proven.

5.1 Theorem: The semaphore invariant method is complete for proving deadlock freedom for those PV programs whose progress graphs do not contain any trap states.

A similar characterization may be given for the progress graphs of those PV programs for which mutual exclusion may be established by the semaphore invariant method. How can the semaphore invariant method be strengthened to handle trap states? Since trap states correspond to "holes" in the unfeasible region of a PV program's progress graph, a method which employs convexity considerations seems promising. We will return to this question again in Section 8.

Although the semaphore invariant method is not complete for proving absence of deadlock, it is a powerful tool for constructing proofs of PV programs which occur in practice. Evidence for the power of the semaphore invariant method may be obtained by comparing it to other methods which have been proposed for proving deadlock freedom of PV programs.

5.2 Theorem: If a PV program has a proof of deadlock freedom using the reduction method of Lipton, then it also has a proof of deadlock freedom using the

semaphore invariant method.

The key lemma used in the proof of the theorem is the following:

5.3 Lemma: If the process graph of a PV program C contains a trap state, then the process graph of the D-reduction C/S also contains a trap state.

6. Generalizations of the semaphore invariant method. Since a large class of synchronization techniques can be modeled by counting operations on shared variables, the class of linear SCL programs is of particular interest. The conditional critical regions of a linear SCL program have the form "with R when $B(v_1, v_2,\ldots v_m)$ do $A(v_1, v_2,\ldots v_m)$ od" where

(1) the variables $v_1, v_2,\ldots v_m$ belong to resource R

(2) the condition $B(v_1, v_2,\ldots v_m)$ is a truth functional combination of atomic formulas of the form $a_1v_1 + a_2v_2 + \ldots + a_mv_m + a_{m+1} \leq 0$

(3) the body $A(v_1, v_2,\ldots v_m)$ is a series of assignment statements which increment the shared variables $v_1,\ldots v_m$, e.g.

$v_1:=v_1 + b_1;\ v_2:=v_2 + b_2;\ldots\ v_m:=v_m + b_m$

Arguments are given in (SC76) and (AG74) that linear SCL programs are universal in their power to express synchronization constraints for concurrent programs.

In a later paper (CL78) we will show how the semaphore invariant of Section 4 can be generalized to linear SCL programs. The generalized semaphore invariant is expressed in terms of a set of formal derivatives $\partial(B_i)/\partial(A_j)$ which express the effect of the execution of the body of the j^{th} critical region on the condition B_i in the i^{th} critical region. Although the generalized semaphore invariant fails to be complete for exactly the same reason as the standard semaphore invariant, it is sufficiently powerful to permit proofs of mutual exclusion and absence of deadlock for a significant class of concurrent programs (e.g. and implementation of the readers and writers problem with writer priority).

7. Fixedpoint techniques for generating resource invariants. Let C be the linear SCL program "$\bar{v}:=\bar{e}$; resource R($\bar{v}$):cobegin $P_1//.../\!/P_n$ coend". For simplicity we assume that C uses only one resource R and that C contains K critical regions for R. We will denote the i^{th} critical region by S_i and assume that it has the form "with R when b_i do A_i od". The canonical annotation described in Section 4 will be used to generate pre and post functions for C. Let the functional F_C be defined by

$$F_C(J) = J_0 \vee \bigvee_{i=1}^{K} SP[A](pre(S_i) \wedge b_i \wedge J)$$

where the predicate $J_0 = \{\bar{v}=\bar{e}\}$ describes the initial state of C.

7.1 Theorem:

(1) Let T be the set of possible states of SCL programs. The functional F_C is a continuous mapping on 2^T. Thus, the least fixedpoint $\mu(F_C)$ of F_C is given by $\mu(F_C) = \bigvee_{j=1}^{\infty} F_C^j(false)$.

(2) All resource invariants IR of C are post-fixedpoints of F_C, i.e. $F_C(IR) \subseteq IR$. Also $\mu(F_C)$ is a resource invariant for C.

(3) The resource invariant system RS_C consisting of $\mu(F_C)$ and the canonical annotation is complete for proving absence of deadlock and mutual exclusion of SCL programs.

(4) If L is a predicate such that $J_0 \rightarrow L$ and $L \rightarrow \mu(F_C)$, then $\mu(F_C) = \bigvee_{j=1}^{\infty} F_C^j(L)$.

Part 4 of Theorem 7.1 is important because it gives a method for generating resource invariants from reasonably close initial approximations. It is analoguous to the use of back substitution for improving invariants of while loops. In a later paper (CL78) we will show how part 4 can be used to generate a resource invariant system for an SCL implementation of the cigarette smoker's problem. We will also show that a number of other techniques for generating resource invariants or verifying concurrent programs can be regarded as special cases of Theorem 7.1. These techniques include the invariant

discovery method described by Keller (KE76), the symbolic evaluation technique of Brand (BR78) and the finite state machine approach of Feldman (FE77).

8. Speeding up the convergence of fixedpoint techniques for approximating resource invariants. The sequence of successive approximation to $\mu(F_C)$ will in general fail to converge unless there are only a finite number of different possible states for the resource R or unless a good initial approximation is available to $\mu(F_C)$. For linear SCL programs the notion of widening developed by Cousot ((CO77), (CO78)) may be used to speed up convergence to $\mu(F_C)$. The widening operator * is characterized by the following two properties: (a) for all admissable predicates U and V, $U \rightarrow U*V$ and $V \rightarrow U*V$ (b) for any infinite ascending chain of admissable predicates $U_0 \subseteq U_1 \subseteq U_2 \subseteq \ldots$ the ascending chain defined by $V_0 = U_0$, $V_{i+1} = V_i * U_{i+1}$ is not strictly increasing.

For our treatment of linear SCL programs, the class admissable predicates will be the polygonal convex sets in Q^m where Q is the set of rational numbers and m is the number of resource variables belonging to R. The widening operator * that we will use is a modification of the one used by Cousot (CO78). Let U and V be polygonal convex sets. Then U and V can be represented as conjunctions:

$$U = \bigwedge_{j=1}^{g_u} a_i^j v_i + a_2^j v_2 \ldots a_m^j v_m + a_{m+1}^j \leq 0, \quad V = \bigwedge_{j+1}^{g_v} b_1^j v_1 + b_1^j v_2 \ldots b_m^j v_m + b_{m+1}^j \leq 0.$$

We will further assume that the representation of U and V is minimal in that no conjunct can be dropped without changing U or V. We will say that two linear inequalities $a_1^j v_1 + \ldots a_m^j v_m + a_{m+1}^j \leq 0$ and $b_1^t v_1 + \ldots b_m^t v_m + b_{m+1}^t \leq 0$ are equivalent if they determine the same half plane in Q^m. U*V will be the conjunction of all those linear forms in the representation of U for which there is an equivalent linear form in the representation of V. Thus the widening operator "throws out" all those constraints in the representation of U which do not also occur in the representation of V and visa versa.

We will now describe our strategy for approximating $\mu(F_C)$. Since the predicates $F_C^j(\text{false})$ in the chain $F_C^0(\text{false}) \subseteq F_C^1(\text{false}) \subseteq \ldots$ are not in general polygonal convex sets, we introduce the functional $G_C(J) = CV[F_C(J)]$ where CV is the convex hull operator. The sequence obtained by iterating G_C, i.e. $G_C^0(\text{false}) \subseteq G_C^1(\text{false}) \subseteq \ldots$ is a chain and each element in the sequence is a polygonal convex set. The sequence I^t, $t \geq 0$ will be used in obtaining a good approximation to the strongest resource invariant for R and is defined as follows:

$$I^t = \bigvee_{j=1}^{\infty} H_j^t \quad \text{where} \quad H_0^t = G_C^t(\text{false}), \; H_{j+1}^t = H_j^t * G_C(H_j^t).$$

Properties of the sequence I^t, $t \geq 0$ are listed in the following theorem:

8.1 Theorem:

(A) Each I^t can be computed in a finite number of steps.

(B) $F_C(I^t) \subseteq I^t$. Thus I^t is a resource invariant for C.

(C) $\mu(F_C) \subseteq I^t$, $t \geq 0$.

(D) The sequence I^t is a decreasing chain in I, i.e. $I^1 \supseteq I^2 \supseteq I^3 \ldots$

Thus the construction of I^t provides an algorithm for computing resource invariants, and this algorithm can be used to obtain successively better approximations to the strongest resource invariant $\mu(F_C)$.

We demonstrate the use of this method of synthesizing resource invariants by considering the program C:

```
a:=0;
cobegin
   P1:cycle produce; V(a) end
   //
   P2:cycle P(a); consume end
coend
```

Note that while the program C quite simple and could in fact be handled by the methods of Section 4, there are potentially an infinite number of states and the method of successive approximations outlined in Section 7 will not

converge. Rewriting the synchronization skeleton of C in terms of conditional critical regions and adding auxiliary variables as described in Section 3, we obtain:

```
a:=0; np:=0, nv:=0;
resource R(a, np, nv):
cobegin
   P1:cycle with R when true do a:=a+1; nv:=nv+1 od end
   //
   P2:cycle with R when a>0 do a:=a-1; np:=np+1 od end
coend
```

Computing the sequence of approximations I^t, $t \geq 0$ we obtain:

I^0 = true

$I^1 = \{a \geq 0 \wedge np \geq 0 \wedge nv \geq 0\}$

$I^2 = \{a \leq nv \wedge np \geq 0 \wedge a \geq 0 \wedge nv \geq 0\}$

$I^3 = \{a+nv-np=0 \wedge np \geq 0 \wedge nv \geq 0 \wedge a \geq 0\}$

$I^3 = I^4 = I^5 = \ldots$

Note that I^3 is exactly the semaphore invariant for the semaphore "a". More substantial examples including the example of Section 5 will be considered in a later paper (CL78).

9. Conclusion. A number of open problems remain regarding the power of the generalized semaphore invariant described in Section 6 and the fixedpoint methods for generating resource invariants described in Sections 7 and 8. In particular it would be interesting to compare these proof techniques with other techniques for proving correctness of concurrent programs which do not use resource invariants, e.g. the Church-Rosser approach of Rosen (RO76) and the reachability tree construction of Keller (KE77). Another open problem concerns the generation of resource invariants for other synchronization methods. Both the discussion of the generalized semaphore invariant and the fixedpoint method for generating resource invariants from linear SCL programs were based on the

assumption that synchronization constraints were expressed as linear inequalities involving resource variables. For high level synchronization methods, such as path expressions (HA75), in which such linear restraints are only implicit, a more direct method of generating resource invariants is desirable.

Finally the author is currently designing an automatic verification system for concurrent programs based on the ideas in this paper. This system will extract the synchronization skeleton of a concurrent program and use the techniques of Sections 6 and 8 to generate the appropriate resource invariants. If this system or some similar system is successful in generating correctness proofs for synchronization skeletons, then a particularly intriguing idea for verifying large concurrent programs would be to combine formal proofs of correctness for the synchronization part of the program with other techniques such as testing or symbolic execution for the sequential part of the program.

References.

(AG74) Agerwala, T. A complete model for representing the coordination of asynchronous processes. John Hopkins University, Baltimore, MD. Computer Research Report 32, 1974.

(BR78) Brand, D. and Joyner, W. H. Verfication of protocols using symbolic execution. Computer Network Protocols. A. Danthine, Editor, University of Liege, 1978.

(CL77) Clarke, E. M. Program invariants as fixedpoints. 18th Annual Symposium of Foundations of Computer Science, Nov. 1977.

(CL78) Clarke, E. M. Synthesis of Resource invariants for concurrent programs. In preparation.

(CO76) Cousot, P. and Cousot, R. Static determination of dynamic properties of programs. Proc. 2nd International Symposium on Programming, B. Robinet, Editor, Dunod, Paris, April 1976.

(CO78) Cousot, P. and Halbwachs, N. Automatic discovery of linear restraints among variables of a program. Proceeding of 5th ACM Symposium on Principles of Programming Languages, 84-96, 1978.

(DE77) DeMillo, M. A., Lipton, R. J., and Perlis, A. J. Social Processes and proofs of theorems and programs. The 4th ACM Symposium on Principles of Programming Languages, January 1977.

(DI67) Dijkstra, E. W. Cooperating sequential processes. Programming Languages, F. Genuys, Editor, Academic Press, New York, 1968.

(FE77) Feldman, Jerome. Synchronization of distant cooperating processes. Computer Science Department Report TR26, University of Rochester.

(FL77) Flon L. and Suzuki, N. Nondeterminism and the correctness of parallel programs. Carnegie Mellon University, Department of Computer Science, May 1977.

(HA72) Habermann, A. N. Synchronization of Communicating processes. Comm. ACM, 15, 3(March 1972), 171-176.

(HA75) Habermann, A. N. Path expresssions. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. June 1975).

(HO69) Hoare, C. A. R. An axiomatic basis for computer programming. Comm. ACM, 12, 10(October 1969), 576-580.

(HO72) Hoare, C. A. R. Towards a theory of parallel programming. Operating Systems Techniques, C. A. R. Hoare, R. H. Perrot, Editors, Academic Press, 1972.

(KE77) Keller, R. M. Generalized petri nets as models for system verification. Computer Science Department Technical Report, University of Utah.

(KE76) Keller, R. M. Formal verification of parallel programs. CACM, 19 (7), 1976.

(LA76) van Lamsveerde, A. and Sintzoff, M. Formal derivation of strongly correct parallel programs. MBLE Research Report, Brussels, Belgium 1976.

(LI75) Lipton, R. J. Reduction: A new method of proving properties of systems of processes. Proceeding of 2nd ACM Symposium on Principles of Programming Languages, 78-86, 1975.

(LM77) Lamport, L. Proving the correctness of multiprocess programs. IEEE Transactions on Software Engineering, 3(2), 1977, 125-143.

(LU72) Lauer, H. C. Correctness in operating systems. Ph.D. Thesis, Carnegie-Mellon University, 1972.

(OW76) Owicki, S. and Gries, D. Verifying properties of parallel programs: An axiomatic approach. CACM 19(5), 279-284, 1976.

(PN77) Pnueli, A. The temporal logic of programs. 18th Annual Symposium on Foundations of Computer Science, November 1977.

(RO76) Rosen, B. K. Correctness of parallel programs: The Church-Rosser Approach. Theoretical Computer Science 2:183-207, 1976.

(SC76) Schmid, H. A. On the efficient implemantation of conditional critical regions and the construction of monitors. Acta Informatica, 6:227-249, 1976.