Computer Science Department

TECHNICAL REPORT

AN APPROACH TO AUTOMATING THE VERIFICATION OF COMPACT PARALLEL COORDINATION PROGRAMS. I

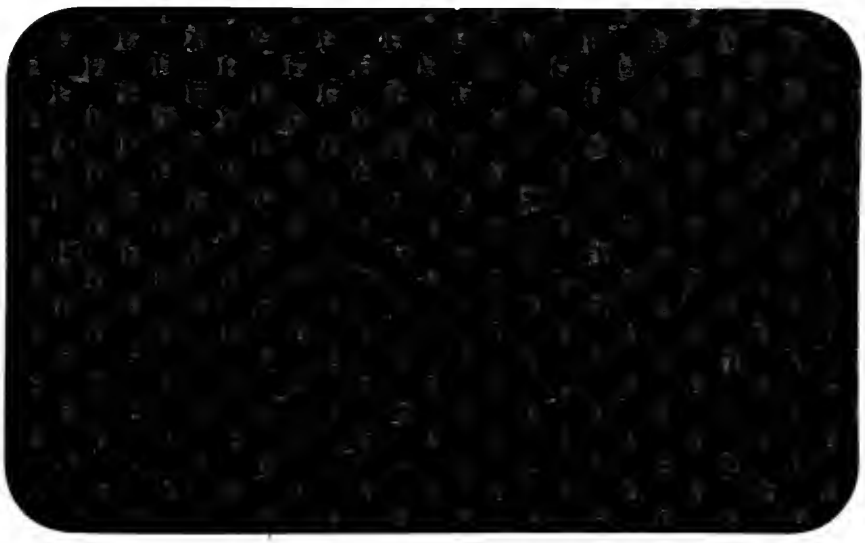
By B. D. Lubachevsky

Computer Science Dept.
New York University
251 Mercer St., N.Y., N.Y. 10012

February, 1983
ULTRACOMPUTER NOTE # 45
Technical Report - # 60

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012



AN APPROACH TO AUTOMATING THE VERIFICATION OF COMPACT PARALLEL COORDINATION PROGRAMS. I

By B. D. Lubachevsky

Computer Science Dept.
New York University
251 Mercer St., N.Y., N.Y. 10012

February, 1983
ULTRACOMPUTER NOTE # 45
Technical Report - # 60

This work was supported in part by the Applied Mathematical Sciences Program of the U.S. Department of Energy under Contract No. DE0AC02-76ER03077, and in part by the National Science Foundation under Grant No. NSF-MCS79-21258.

Ultracomputer Note #45 August 1982

AN APPROACH TO AUTOMATING THE VERIFICATION OF COMPACT PARALLEL COORDINATION PROGRAMS. I

B. D. Lubachevsky

Abstract - A class of parallel coordination programs for a shared memory asynchronous parallel processor is considered. They use the primitive operation Fetch&Add and differ in various respects from those which use basic P and V operations. The F&A is the basic primitive for the "NYU-Ultracomputer". A correctness proof for the considered programs must be done for arbitrary number N of processing elements since the Ultracomputer design includes thousands of PEs.

A so-called reachability set description (RSD) is introduced, in which all reachable states of a program (exponential function of N) are collapsed into a fixed number of "metastates". The transitions between these "metastates" are also specified. By using such a notation it becomes feasible to prove the absence of livelock and other temporal properties of a parallel program. The concept of a compact parallel program is introduced. Roughly speaking a parallel program executed by N PEs is compact if there exists a boundary T independent of N such that any state of this program is reachable within time T. Compactness may also be understood as the uniform over N finiteness of the expansion for the strongest invariant in the least fixpoint theory. A program that builds RSDs for compact parallel programs and examples of proofs generated by this program are discussed.

Typographical conventions

Due to a limited type font on the device that printed this manuscript, the following conventions are used:

Symbol in the paper	Usual shape	Meaning	Example of use
Ω	$\cap$	set intersection	A Ω B
<<	$\subset$	set containment	A << B
not<<	$\not\subset$	negation of set containment	A not<< B
>>	$\supset$	set inclusion	A >> B
not>>	$\not\supset$	negation of set inclusion	A not>> B
≤≤	$\in$	element to set containment	a ≤≤ B
not≤≤	$\notin$	negation of element to set containment	a not≤≤ B
≥≥	$\ni$	set to element inclusion	A ≥≥ b
not≥≥	$\not\ni$	negation of set to element inclusion	A not≥≥ b

Contents

1. Introduction.

The programs developed for the "NYU-Ultracomputer", a shared memory asynchronous parallel processor [6], are characterized by the following two properties: 1) they are to be executed by a large system (the parallel computer might include thousands of processing elements); 2) they use the basic primitive Fetch&Add which enables efficient parallel programs to be written for such systems [5, 17].

As a price for the efficiency of a F&A-program the possibility of making a specific time-related mistake in such a program is seemingly greater when comparing it not only with a serial program (which is not surprising) but even when comparing it with a conventional PV-program of the same length. (In a PV-program the only coordination primitives are P and V operations on semaphores.)

The difference in the complexity of a F&A-program and a PV-program manifests itself in the fact that the skeleton of a PV-program is equivalent to both a Petri Net and to a vector-addition system (VAS) [7, 9], whereas the skeleton of a F&A-program is generally not equivalent to a Petri Net, and a generalization of the notion of VAS is required for its description (such a generalization, called controlled VAS, will be described in part II of this paper).

A skeleton of the "naive" semaphore (see section 2.2) is a classical example of the complexity of a F&A-program. These 3 lines of code (each line with one assignment) contain an inadmissible livelock which is not readily discernible[1].

We have tried to apply the assertional approach [1, 11, 13, 14] to automating the debugging and correctness proof for the considered class of programs. In this approach an assertion about the program state (augmented by a number of auxiliary variables when necessary) must be supplied. From this assertion desired properties of the program are supposed to follow. The predicate that expresses the assertion must be proven true for the initial state and the true value of this predicate must be proven invariant for any program transition. If such a proof holds then this predicate is true for any reachable state of the program and it is called an invariant of the program.

The task of proving the invariance of a given assertion does not present a difficulty for all the considered examples. The task of supplying the auxiliary variables and of generating the invariant appears to be more difficult. A method for the invariant generation,

[1] The livelock was not recognized by anyone shown the program for the first time by the author. This bug was exposed by Dijkstra ([4], p. 122). Since on basis of this livelock existance primitive F&A was rejected in [4] as unappropriate to solve the mutual exclusion problem (swap-operation was suggested instead), Dijkstra seemingly did not realize the fact that only one extra checking statement eliminates the livelock (see section 3). All these facts confirm the point that it is difficult to obtain an "insight" of a F&A-program. Note that this difficulty is only inherent to coordination routines, written in an unstructured F&A-style. The latter take a small fraction of an application code, but an essential part of the operating system.

suggested in [8] for PV-programs, can be generalized to include F&A-programs. However, the invariants supplied by this method are not strong enough to prove, for example, the absence of a deadlock in some cases [2].

There is yet another difficulty in using the assertional method for F&A-programs. Namely, this method (at least as presented in [13, 14]) does not help to prove the absence of blocking for some of these programs. (See section 3.4 for an example.)

Facing these difficulties the author decided to examine a method based on an explicit presentation of the graph Δ of all the program states and state transitions. Generally large size of Δ is considered to be the major obstacle in using this method. The size of Δ depends on the length of the analysed program and on the number N of executing processing elements (PEs). Although the length of the considered F&A-programs is small, the size of Δ grows very quickly with N.

It was observed, however, that in most cases it is not essential which PE executes which statement but rather how many PEs execute each statement. The consequent symmetrization thus reduces the growth of Δ from exponential with respect to N to polynomial with respect to N.

A program, called verifier-1 (see section 2), was created that capitalizes on these advantages. Verifier-1 first builds the (symmetrized) description of the reachability graph and then answers various questions about this graph.

The general types of questions which appear useful are:

Is there a reachable state satisfying a given predicate?

Is there a strongly-connected component of reachable states satisfying a given predicate and a special infinite loop property? (This property relates to the well-known finite delay and fairness properties. The discussion follows in section 2.11.)

Verifier-1 builds Δ for fixed moderate values of N. The correctness of a property established by verifier-1 does not automatically mean that this property is true for arbitrary N. (The case of interest for the Ultracomputer.)

To verify correctness for arbitrary N another program, called verifier-2, was created. Verifier-2, if terminated, produces the so-called reachability set description (RSD) for the analysed parallel program. In the RSD graph Δ is represented in a form even more aggregated than the symmetrized form produced by verifier-1. All reachable states of the analysed program are collapsed into a fixed number of the so-called metastates. Each metastate is a set in the state-space of the program, represented by a conjunction of a fixed number of inequalities, which are linear with respect to the state-space coordinates. The transitions of the analysed program are also represented in an aggregated form as transitions between these metastates.

Since all the states represented in the union of these metastates are reachable, the RSD serves as a solution to the problem of invariant generation. Namely, one can view the disjunction of descriptions of the metastates as the strongest invariant for the analysed program. Moreover, proving the absence of blocking becomes feasible in the RSD notation. Namely, from the RSD one can produce a compact description of all strongly-connected components in Δ satisfying a given predicate, then by studying these components one can check if any of them generate a blocking or not. Examples in sections 3 and 4 demonstrate such a verification scheme.

A necessary condition for the termination of verifier-2 when building a RSD for a parallel program is the compactness of this program. Compactness with respect to the set V_0 of initial states of the program means that there exists a boundary T_0 (independent of N) such that any state is reachable from a state of V_0 within time T_0.

Compactness may also be understood in terms of the expansion for the strongest invariant in the least fixed-point theory [2]. This expansion has the form

$$R(V_0) = \bigcup_{j \geq 0} F^j(V_0), \tag{1.1}$$

where $R(V_0)$ is the reachability set of the program or else the carrier of the strongest invariant; F is the progress functional which can be obtained from the text of the program (see section 2.9) $F^{j+1}(x) \overset{Df}{==} F(F^j(x))$, $F^0(x) \overset{Df}{==} x$.

Compactness requires that the expansion (1.1) contain a fixed (finite) number of terms independent of N.

Normal programs, a restricted class of parallel programs (that still includes many interesting examples) are specified in section 4. It will be proven in part II that if a program is normal then compactness is sufficient for verifier-2 to produce the RSD in a finite time (if enough memory is available to the verifier).

The example in section 2.8 shows that the verifiers do not only work for F&A-programs. Programs using P and V operations on semaphores may be verified as well. However, the following restrictions on the analysed program must be satisfied even when using the less restrictive verifier-1:

(v1) the code of the analysed program must be the same for all N processing elements (PEs);

(v2) private variables for each PE may take only a fixed number of different values independent of N;

(v3) there is only a fixed number of public variables each of which takes a finite number of values (the latter number may depend on N).

Property (v1) seems to be least restrictive, whereas properties (v2) and (v3) are restrictive. In particular, they infer finiteness of the reachability graph for any given N.

Despite these restrictions the author tends to consider these verifiers to be successful. For some of the programs verified by this method, no other formal proof technique known to the author seems to work.[2]

In this, part I of the paper, an introduction to this method is given. Several examples of its application are demonstrated.

2. Verifier-1.

In this section the theoretical background of the algorithm, called verifier-1, is described[3]. We first briefly discuss the Ultracomputer (for a more extended discussion see [6, 18]). Then the work of verifier-1 is demonstrated in the verification of a simple but an instructive F&A-program. A general class of parallel programs to which this verifier may be applied is then introduced and an example of the verification of a non-F&A-program is presented. In discussing these examples, several notions relevant to verification are introduced. Only those constructions and notions introduced in sections 2.5, 2.7, 2.13, and, possibly, 2.12 seem to be novel. The

[2]Although the temporal logic approach [15, 16, 18] may help in studying various properties of the considered programs for fixed moderate values of N, this approach does not address the case in which N is a parameter. In general going from N = 2 to arbitrary N is non-trivial for F&A-programs.

[3]Verifier-1 has been implemented as a FORTRAN program. We will not discuss aspects of its programming implementation since they are obvious.

others are known from elsewhere and a discussion of them is included to faciliate reading by someone not familiar with the subject.

2.1. Ultracomputer and primitive Fetch&Add.

An ideal parallel processor, dubbed a "paracomputer" by Schwartz [18], consists of identical PEs sharing a common memory. The individual PEs may also have attached local memory, which we refer to as their "private" memories; the memory shared by and common to all processors is called "public", and variables stored there are called "public variables". The PEs can simultaneously read any public cell in one cycle. Moreover, simultaneous writes (including the F&A operation) are likewise effected in a single cycle and a memory cell to which such writes are directed will contain some one of the quantities written into it. Note that simultaneous memory updates are not serialized; in fact they are accomplished in one cycle.

The programmer may view the Ultracomputer as a paracomputer and we will treat the Ultracomputer this way in our subsequent consideration[4].

[4]Paracomputers must be regarded as idealized computational models since physical fan-in limitations prevent their realization. Ultracomputer is a realizable approximation to a paracomputer in which each PE can directly access its private memory and can access the public memory via a (multicycle) interconnection network. In this more realistic architecture a public memory access may require several PE cycles.

The code of an Ultracomputer program looks like one of an ordinary serial program. Each PE "sees" the public memory and executes the program code. In this paper only the case when this code is identical for all the PEs is considered. The speed of these executions may be different for different PEs. Fetch&Add is the primitive to coordinate PEs.

The format of the F&A operation is F&A(c,i), where c is a variable and i is an expression. This indivisible operation yields c as its value and replaces the contents of storage location c by c + i. In this paper only the case when variable c is public integer and i is an integer constant is considered.

The following principle of serialization of operations F&A takes place:

Assume that several (possibly very many) F&A operations simultaneously address c. Then the effect is as though these operations occurred in some (unspecified) serial order, i.e. c receives the appropriate total increment and each operation yields the intermediate value of c corresponding to its position in this order.

For example, if PE1 executes $p_1 \leftarrow$ F&A(c,i_1), and if PE2 simultaneously executes $p_2 \leftarrow$ F&A(c,i_2), and if c is not simultaneously updated by another PEk ($k \neq 1,2$), then either $p_1 \leftarrow c$, $p_2 \leftarrow c + i_1$, or $p_1 \leftarrow c + i_2$, $p_2 \leftarrow c$, and, in either case, the value of c becomes $c + i_1 + i_2$. The first possibility corresponds to the serialized order

in which PE1 executes its F&A and then PE2 executes its F&A; the second possibility corresponds to the opposite serialization.

It is also possible to have loads, stores, and F&As all concurrently directed at the same memory location. The effect is as though these operations occurred in some serial order. (In [5, 6] a hardware design is presented in which the F&A operation requires essentially the same execution time as a load or store and in which many simultaneous F&As updating the same variable are processed particularly efficiently.)

2.2. "Naive" semaphore.

Following [4] we will try to write a F&A-program which provides a mutual exclusion. More specifically, our parallel program has the form:

Program 2.1

```
cycle: entry
       {critical section; executed for a finite time}
       exit
       {remainder of cycle; executed for a finite time}
```

Codes for "entry" and "exit" ought to be programmed so that the following two properties are satisfied:

(CSe) No more than 1 PE can be in the critical section (CS) at any one time.

(CSr) For any time t there exists a time $t' \geq t$ such that some PE is in the CS at time t'.

Above CSe stands for Critical-Section-exclusion, CSr stands for Critical-Section-reachability. (Note that we are not concerned here about the possible starvation or busy-waiting of a particular PE.)

The Program 2.2 is a "naive" solution to the above problem, since it contains a blocking. We will use this example to demonstrate the work of verifier-1.

Program 2.2

```
COMMENT sem is public; initially sem = 1;
        no {...} section accesses sem

P1:  if  F&A(sem,-1) ≥ 1  then go to P3   |
                                          |   entry
P2:  F&A(sem,1)                           |
     go to P1                             |

P3:  {critical section; executed for a finite time}

     F&A(sem,1)                           |   exit

     {remainder of cycle; executed for a finite time}
     go to P1
```

Assuming[5] that both sections enclosed in {...} in Program 2.2 are basic blocks, i.e. each has a single entry and a single exit, we will treat these sections as null statements. Thus, we consider the skeleton Program 2.3.

Program 2.3

```
COMMENT sem is public; initially sem = 1.

     P1:  if  F&A(sem,-1) ≥ 1  then go to P3

     P2:  F&A(sem,1)
          go to P1

     P3:  F&A(sem,1)
          go to P1
```

2.3. Reachability graph. Example of the "naive" semaphore.

The code of the skeleton of an analysed program and a number N of PEs in the Ultracomputer are inputs to the verifier-1. For Program 2.3 and N = 2 the verifier produced the output shown in Fig. 2.1.

In Fig. 2.1 all states $s_1,\ldots,s_5$ of the program are presented, one state per row. Each state s is characterized by four integer numbers, $s = (n_1, n_2, n_3; sem)$, where n_i is the number of PEs at position Pi, sem is the corresponding value of the public variable. For example, $s_1 = (2,0,0;1)$. For each state all possible state transitions when one PE is moving are specified. For example at state s_2 a PE may move from

---

[5] The code of "entry" in Program 2.2 might be rewritten in a more usual "while"-style without explicit labeling:

```
while (F&A(sem,-1) < 1)
       F&A(sem,1)
```

For the purposes of verification we expose labels in the "goto"-style as in Program 2.2 and even insert the unreferenced label P2. Section 2.4 explains the reason for this insertion.

two positions: P1 and P3. Hence two transitions are possible. One is generated by a PE moving from P3 to P1, the corresponding entry in the table is "P3->P1". This transition leads to state s_1. The other is generated by a PE moving from P1 to P2, the corresponding entry in the table is "P1->P2". This transition leads to state s_3.

Fig. 2.1 describes the reachability graph Δ of the program, i.e. the graph of all program states and state transitions (Fig. 2.2). A state in Fig. 2.1 corresponds to a node in Δ and a transition in Fig. 2.1 corresponds to an arc in Δ.

	states				state transitions				
	P1	P2	P3	sem	s_1	s_2	s_3	s_4	s_5
s_1	2	0	0	1		P1->P3			
s_2	1	0	1	0	P3->P1		P1->P2		
s_3	0	1	1	-1		P2->P1		P3->P1	
s_4	1	1	0	0	P2->P1				P1->P2
s_5	0	2	0	-1				P2->P1	

Fig 2.1. Table representation of the reachability graph for Program 2.2.

2.4. Indivisible operations and the serialization principle.

Observe that each statement labeled Pi in Program 2.3 contains no more than one public memory reference. This demonstrates a general rule: labels Pi must be inserted so that each statement Pi contains at

most one public memory reference. (This is the reason to insert an unreferenced label P2 in Program 2.2.)

Let a program be written and labels Pi be inserted according to this rule. Then the verifier will ignore states in which a PE has executed part, but not all, of a statement Pi. In other words, statements Pi are considered as indivisible (i.e. atomic) operations.

For example, if V is a public variable, and P is a private variable, then statements such as V <- 1, P <- V, F&A(V,P) are considered as indivisible operations whereas statements such as V <- f(V), F&A(V,-V) are not. In particular, since execution of F&A(V,-V) might be interleaved with other statements which reference location V, F&A(V,-V) might not yield 0 as the computed value of V.

Now we formulate the so-called serialization principle:

Suppose a program is partitioned into atomic operations. (These operations may be high level language statements as statements Pi in Program 2.3; or they may have finer granularity, e.g. machine instructions may be considered as atomic operations.) Then effect of an execution of the program is the same as if each PE executed one atomic operation at a time and the executions of these atomic operations formed some (unspecified) serial order.

As mentioned above, we assume that each statement Pi which contains at most one public memory reference may be treated as an

atomic operation. This implies the "statement-level" serialization principle.

The notion of the "effect of execution" will be used unformally, hence no rigorous proof of the "statement-level" serialization principle will be presented. The following heuristic argument suggests its validity by reducing it to the "instruction-level" serialization principle.

Suppose that "effect of execution" is defined somehow (in accordance with the common sense) and that the "instruction-level" serialization principle is guaranteed. Let γ = ...Ik, I(k+1),... be the sequence of indivisible machine instructions (each instruction is executed by one PE), such that the "effect" of serial execution of γ is the same as one of the original parallel program. Such a γ exists by virtue of the "instruction-level" serialization principle. (If the definition of the "effect of execution" is "good", then) it must be clear that without changing this "effect" one can interchange any two neighbors in γ providing that at most one of them references a public variable. There exists a sequence of such interchanges that yields an execution order in which each statement Pi is indivisibly executed.

2.5. Bose and Boltzmann semantics.

The representation of Δ in Fig. 2.1 is symmetrized in the following sense. A node in Δ indicates how many PEs are at each position Pi in the program but it includes no identification of these

PEs. In statistical physics there exist Bose statistics and Boltzmann statistics to describe ensembles of particles. The difference between these two statistics is similar to the difference between the two ways of representing a state of a parallel program. Bose statistics corresponds to the symmetrized description, Boltzmann statistics corresponds to the other (refined) description. We will use the same nomenclature with respect to the program states.

Programs we consider do not include a PE identification. Such a program can not distinguish PEs. During execution it may assign an identification to the process carried by a PE. For example in Program 2.2 one may speak of a PE that is in the CS but there is no way to find out which PE this is. Note that the representation of Δ in Bose semantics may hide the so-called starvation of an individual PE, when this PE can never get access to some program section. We do not care about this kind of starvation.[6] The starvation of a process is only of interest to us and it may be identified in Bose semantics if one separates the program section under question by replicating the code of this section as discussed in section 2.12.

[6] In fact, even Program 3.1 which will be considered in section 3.1 as a solution to the mutual exclusion problem satisfying properties (CSe) and (CSr) can starve a PE and hence its process. However, there are many applications in which the possibility of such starvation is not dangerous. An efficient starvation-free F&A solution of the mutual exclusion problem can be built as well [17].

We now give a rough estimation of the complexity of Δ under the two ways of representation of Δ. The number of all possible distributions of N "particles" (PEs) among k possible "states" (statements of a program) in "Boltzmann statistics" is k^N or else the sum of terms $N!/(n_1!...n_k!)$ extended to all (integer non-negative) solutions of the equation

$$n_1 +...+ n_k = N, \tag{2.1}$$

i.e. it is exponential with respect to N, given a fixed k (size of the program). In "Bose statistics" (with indistinguishable particles) all states corresponding to a solution (2.1) reduce to a single state. Hence their total becomes number of different solutions of (2.1), i.e.

$$\frac{(N + k - 1)!}{N! \times (k - 1)!} = \frac{N^{k-1}}{(k - 1)!} + O(N^{k-2}).$$

This is a polynomial with respect to N given a fixed k.

It will be shown in section 2.10 that graph Δ for Program 2.3 has $2N + 1$ states in Bose representation and $N \times 2^{N-1} + 2^N$ states in Boltzmann representation.

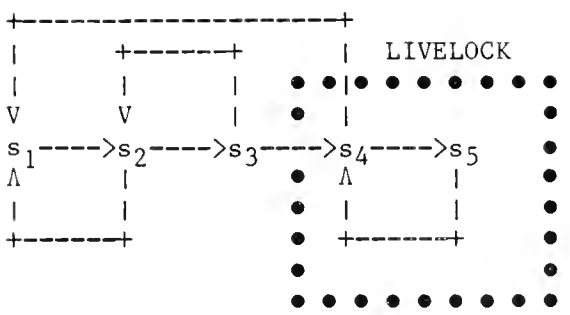


Fig. 2.2. Reachability graph for Program 2.2 and its livelock.

2.6. Livelock.

Given Fig. 2.1 verifier-1 checks property (CSe) (and finds it true) by examining all states $s_1,\ldots,s_5$.

To check property (CSr), a subgraph of Δ generated by the predicate $n_3 = 0$ is examined. In this subgraph only the cycle s_4->s_5->s_4 exists. This cycle is produced by verifier-1 as a possible scenario of blocking in which no PE can access statement P3 (in the skeleton Program 2.3 statement P3 represents a CS of the concrete Program 2.2). In this cycle each of the two PEs executes loop P1->P2->P1 and prevents the other PE from entering the CS. This situation may regenerate itself infinitely many times.

Such blocking is similar to a deadlock. The difference between this blocking and a deadlock is the following. There exists a possibility to quit this blocking cycle (from state s_4 to state s_1 if one PE slows down its execution at P1), whereas in the deadlock type of

blocking there is no way to quit without terminating execution. Such blocking is called a livelock in the existing literature (see [10]).

For natural assumptions about the distribution of execution times it can be shown that as N increases the probability to enter this livelock increases and the probability to quit it decreases. In practice this livelock for large N is equivalent to a deadlock and the suggested solution to the mutual exclusion problem is not acceptable.

2.7. Fetch&Function-programs.

A general skeleton-program for which verifier-1 can build the reachability graph is defined by the following:

- a directed graph G (which represents the "body" of the analysed program), the vertices of which are called positions. Let {Pi| i = 1,...,k} be the set of positions of G; G has two kinds of positions: interior and exterior, described below. As usual we say that Pj follows Pi, if G has an arc (Pi,Pj);

- a set {c} of counters, such that to each interior position Pi there is associated a counter c = c_j, so that j = j(Pi). Note that one counter c_j can be associated with several Pi's. Set {c} corresponds to the public variables in the concrete program. The analysed programs are supposed to have no private variables. (A program with a limited number of bounded private variables may be reduced in this manner as

will be shown in section 2.12.) Therefore {c} coincides with the set of all program variables;

- a set $\{d_i\}$ of directing functions; d_i assigns to each value of $c_{j(Pi)}$ a position that follows Pi. Each interior position has its own directing function. If no position follows Pi then $d_i(c_{j(Pi)})$ is the empty set.

- a set $\{f_i\}$ of replacing functions, f_i maps Range($c_{j(Pi)}$) to itself. Each interior position has a unique replacing function.

The distinction between interior and exterior positions is that the former usually represent the control flow of a concrete program represented by the skeleton program; whereas the latter represent the control flow of other "exterior" concrete programs. (Program 2.3 has no exterior positions, examples of exterior positions will be supplied by programs below.)

Now we wish to present our skeleton program in a form closer to the usual representation of a concrete program. We will use the expression F&f(c) for the function or procedure (this depends on context) with the side-effect that fetches the old value of c and replaces c by f(c). In this new representation, the program will be written as an unordered set of statements each of which corresponds to a position of G. There are two kinds of statements: for each interior position Pi the corresponding statement is

(I) Pi: go to $d_i(F\&f_i(c_{j(Pi)}))$,

for each exterior position Pi the corresponding statement is

(E) Pi: go to Pj_1 or Pj_2 or...or Pj_m,

where $Pj_1,\ldots,Pj_m$ follow Pi. The expression "go to empty", that occurs when no position follows Pi, should be understood as the empty statement "do nothing".

In the example programs we consider, the statement order is significant because we adopt the usual convention that if no "go to" appears in the statement corresponding to Pi, a "go to the statement corresponding to P(i+1)" is assumed. The following statement forms (all are special cases of (I) and (E)) are used in the examples.

(I1) Pi: counter <- constant

(I2) Pi: counter <- constant
go to Pj

(I3) Pi: if predicate(counter) then go to Pj

(I4) Pi: if predicate(counter) then go to Pj_1
go to Pj_2

(I5) Pi: $F\&f_j$(counter,constant)

(I6) Pi: $F\&f_j$(counter,constant)
go to Pj

(I7) Pi: if predicate($F\&f_j$(counter,constant)) then go to Pj_1

(I8) Pi: if predicate($F\&f_j$(counter,constant)) then go to Pj_1
go to Pj_2

(E1) Pi: go to Pj

(E2) Pi: go to Pj_1 or Pj_2

2.8. Example of verification of a program with P and V operations.

Consider the problem of simulating a general semaphore using binary semaphores. More specifically "entry" and "exit" routines for Program 2.1 have to be written so that at most K PEs may stay in the CS. Here K is a positive integer parameter. The only operations referencing common memory may be read, write and also P and V operations on binary semaphores.

Program 2.4 is a solution for this problem proposed in [20], p. 78. Note that in the subsequent edition of this book the program was modified. Our code is close to the original ALGOL-like program. Labels Pi are inserted according to the rule in section 2.4. Note that we treat statements g <- g-1 and g <- g+1 as indivisible. Their indivisibility must be proven first, and it can be done using the resolution rule discussed in section 2.13. However we need not present the proof since we are going to expose a bug in this program.

Program 2.4

```
COMMENT g, m, d are public; m stands for mutex; d stands for delay;
        g simulates a general semaphore variable, integer;
        g can take negative values (unlike a genuine semaphore variable);
        m and d are binary semaphores, i.e. they take only values 0 or 1;
        initially g = K, m = 1, d = 0;
        no {...} section accesses g, m or d.

P1:  P(m); P2: g <- g-1; P3: if  g ≤ -1  then                    |
                             begin P4: V(m); P5: P(d) end        | entry
                             else  P6: V(m);                     |

P7:  {critical section; executed for a finite time}

     P(m); P8: g <- g+1; P9: if  g ≤ 0  then P10: V(d);          | exit
           P11: V(m);                                            |

     {remainder of cycle; executed for a finite time}
         go to P1
```

The Fetch&Function model of section 2.7 can represent this program. To follow the Fetch&Function style one should replace each occurrence of P(semaphore) and V(semaphore), respectively, by statements

(2.2p) Pi: if F&f (semaphore, -1) ≤ 0 then go to Pi

(2.2v) Pi: F&f (semaphore, 1)

where the function f is defined by the formula f (semaphore, increment) = min (max (semaphore + increment, 0), 1).

Verifying this program, we pick values N = 3, K = 2. Verifier-1 first produces the reachability graph Δ. Δ contains 61 nodes. Then verifier-1 is asked whether there are states violating property (CSe) of section 2.2. No such state is found. Then it is asked to produce a loop which generates blocking like the one in section 2.6 (when all PEs

execute but no one can get into the CS). No such loop is found. Thus, property (CSr) of section 2.2 is confirmed.

A general semaphore program must also possess the following general reachability property:

(CSGr) For any $k < K$, if k PEs are in the CS, then any number up to K-k PEs which start executing the entry-section will eventually enter the CS, and the delay of these PEs will not depend on how long those k PEs remain in the CS.

Note that (CSr) in section 2.2 may be considered as a particular case of (CSGr) when $K = 1$, and $k = 0$. To check (CSGr) the following question was asked: is there a loop wherein one PE is in the CS and the other two PEs execute but can not get into the CS? To answer this question verifier-1 examines the subgraph Δ' generated in Δ by the predicate $\{n_7 = 1\}$. Then verifier-1 produces a loop in Δ' consisting of a single blocking state s_b. In s_b one PE is in the CS (position P7 of the program) and the other two PEs are waiting at the closed semaphore d at P5: $s_b = \{n_5 = 2, n_7 = 1, g = -1, m = 1, d = 0\}$. (More accurately, the two PEs at P5 are not waiting but are executing a busy-waiting loop like (2.2p).)

Then verifier-1 is asked to produce a scenario of this bug, i.e. a path in Δ starting from the initial state in which all three PEs are at P1 and terminating at s_b. This scenario consists of 21 steps, where one step corresponds to a PE completing execution of a statement Pi.

Here are these steps:

At first, (steps 1 to 4) one PE is advanced to P7. Then (steps 5 to 8) another PE is advanced to P7. Now g = 0, m = 1, d = 0. When the last PE tries to gain access to the CS it is forced to stop at statement P5 (steps 9 to 12), since at P3 the value of g is -1. One PE then leaves the CS and returns to starting position P1 (steps 13 to 17). Now the general semaphore g is "closed", i.e. g = 0, one PE is at P1, one PE is at P5, one PE is at P7, m = 1, d = 0. Then the PE at P1 tries to get to the CS for the second time (steps 18 to 21) but is forced to stop at P5 after step 21, since at P3 the value of g is -1. Thus we obtain the blocking state s_b.

In the following table number of states reachable by the program is given for various N and K. This table demonstrates a Δ of moderate complexity for small N.

\N K\	1	2	3	4	5
1	8	22	35	48	61
2	8	15	61	90	119
3	8	15	22	114	159
4	8	15	22	29	181
5	8	15	22	29	36

2.9. Reachability graph and progress functional.

In developing the reachability graph verifier-1 generates states of the analysed program in Bose semantics. In these semantics the

execution state (or simply the state) of a program is declared as the vector

$$s = (n_1,\ldots,n_k;\ c_1,\ldots,c_r), \tag{2.3}$$

where n_i is the current number of PEs at position Pi, $i = 1,\ldots,k$; $N = n_1 +\ldots+ n_k$ is the total number of PEs; c_i is a counter; and r is the number of distinct counters.

As it follows from section 2.7 that to know the text of the program without private variables is the same as to know the set $C = (G, \{c\}, \{f_i\}, \{d_i\})$, we will not distinguish between them.

Given a program text C we define the progress functional $F = F_C$ as follow. If s is a program state as in equ. (2.3), then F(s) denotes the set of all states reachable from s when each PE executes at most one statement. If argument of F is a set $\bigcup_i \{s_i\}$ of program states, then the value of F is defined as $F(\bigcup_i \{s_i\}) \overset{Df}{==} \bigcup_i F(s_i)$.

For Program 2.3, whose five states are presented in Fig. 2.1, one has:

$F(s_1) = \{s_1, s_2, s_3\}$, $F(s_2) = \{s_1, s_2, s_3, s_4\}$,

$F(s_3) = \{s_1, s_2, s_3, s_4\}$, $F(s_4) = \{s_1, s_2, s_4, s_5\}$, $F(s_5) = \{s_4, s_5\}$.

Note that F(s) always includes s, all states that directly follow s in Δ, and it may include some other states. For example, s_3 does not directly follow s_1 in graph Δ. However, $F(s_1)$ contains s_3, since moves

of one PE from P1 to P3 and of the other PE from P1 to P2 generate s_3. Although these moves are simultaneous one may serialize them so that the latter move follows the former.

Starting with a given initial state s_1 of Program 2.3 the verifier generates a monotonically increasing sequence of sets $F(s_1)$, $F^2(s_1)$, $F^3(s_1)$, and terminates after it recognizes that this sequence stabilizes at $F^3(s_1)$ (i.e. $F^3(s_1) = F^4(s_1) = F^5(s_1) = \ldots$). $R(s_1) \overset{Df}{==} F^3(s_1)$ is the reachability set of this program.

Though the term "reachability set" has been used several times in this paper, it has not been yet formally defined. Now we can do this since the necessary preliminary definitions have been given.

Let V_0 be a set of states (2.1) of a skeleton program C. The reachability set $R=R(V_0)$ for the set of initial states V_0 is defined either by expression (1.1) or by $R(V_0) \overset{Df}{==} \underset{j->}{\text{limit}}\ F^j(V_0)$ (since $F^j(V_0)$ monotonically increases with j). All elements of $R(V_0)$ are called reachable (from V_0) states.

Note that sequence $F^i(V_0)$, i = 1,2,... might not stabilize. For example, this sequence will not stabilize for Program 2.5 when V_0 is finite.

Program 2.5

```
P1: F&A(c,1)
    go to P1
```

In the case when V_0 is finite (in particular, if V_0 reduces to a single initial state s_1) sequence $F^i(V_0)$ stabilizes if and only if graph Δ is finite. Graph Δ is finite if and only if all counters c_i may take only a finite number of values. This is property (v3) of the introduction.

In particular, if all c_i's are integers then the latter condition is equivalent to the boundedness of the counters. When working only with integer counters, bounds on the counters are inputs for verifier-1. They must be supplied by the programmer.

If when developing the reachability set a state s_* which violates these bounds is generated, then the process of generating the reachability set terminates. As a diagnostic, verifier-1 produces a sequence of states beginning with one of the initial states and leading to the state s_*. Note that bounds on the counters might constitute one of the possible expectations of the programmer about the program behavior. Hence their violation might be interpreted as a program bug.

The reachability directed graph $\Delta = \Delta(V_0)$ is defined as follows: the node set of Δ is $R(V_0)$ and the arcs are the possible transitions, i.e. all pairs of reachable states (s,s') such that s' may be obtained from s by moving one PE from the statement Pi to the corresponding next statement Pi'.

2.10. Invariants and the least fixed-point theory.

Let V_0 be a set of states (2.3) of a F&f-program and pr(s) be a predicate on program state s. We call pr(s) a program invariant with initial states drawn from the set V_0 (or simply an invariant) iff the carrier S of pr(s), $S = carrier(pr) \overset{Df}{==} \{s \mid pr(s) = true\}$, satisfies the properties

(2.4) $$F(S) << S,\ V_0 << S.$$

It is always true that $S << F(S)$ (because progress functionals include the identity mapping, which corresponds to no progress). Therefore the first relation (2.4) may be replaced by equality: $F(S) = S$.

For example, the following predicate $pr_1(s)$ is an invariant for Program 2.2 with initial states s drawn from the set $V_0 = \{s = (n_1,n_2,n_3;sem) \mid n_2 = n_3 = 0,\ sem = 1\}$:

(2.5) $$pr_1(s) = \{sem = 1 - n_2 - n_3\}.$$

(We imply that $n_i \geq 0$, $i = 1, 2, 3$ and that n_i and sem are integers.) To prove this, one takes arbitrary state $s = \{n_1, n_2, n_3; sem\}$ satisfying pr_1 and verifies the fact that every PE move generates a state satisfying pr_1.

If set S satisfies condition F(S) = S then it is called a fixed-point of the functional F. We will only consider fixed-points S satisfying the additional condition $V_0 \ll S$ and will not distinguish between fixed-points of the progress functional and the program invariants. Since F(S) monotonically increases with S, if sets S_1 and S_2 are fixed-points of F, then $S_1 \Omega S_2$ is a fixed-point of F. Therefore, given V_0 the strongest invariant exists uniquely and is the conjunction of all possible invariants. The carrier of this invariant is the least fixed-point of F.

The following predicate pr_* represents the strongest invariant for Program 2.3 with the single initial state $s_1 = (N, 0, 0; 1)$, N = 1,2...:

$$(2.6) \quad pr_*(s) = \{s \mid sem = 1-n_2-n_3,\ n_1+n_2+n_3 = N,\ n_3 \leq 1\}.$$

N in equ. (2.6) is the total number of PEs. Both expressions for the size of Δ given in section 2.5, polynomial and exponential, follow from equ. (2.6). For a fixed value of N equation (2.6) is not a unique expression for pr_*. For example if N = 2 then another valid expression is:

$$(2.7) \quad pr_*(s) = \{s = s_1\} \text{ OR...OR } \{s = s_5\},$$

where the s_i are states of Program. 2.1 as in Fig. 2.1.

In this example, the carrier of the strongest invariant is equal to the reachability set. This is true in general case [2].

Comparing (2.6) with (2.7) we indicate that expression (2.6) has an obvious advantage over (2.7) in that the length of representation (2.6) is independent of N. A mechanical way of producing expressions like (2.6) will be discussed in section 3.

What is usually implied by the term "program invariant" is a compact presentation, like equ. (2.6) in our example. However one should realize that for finite fixed reachability sets there is no formal difference between these two presentations.

2.11. Infinite loop property.

Cycle $T = \{s_4 \rightarrow s_5 \rightarrow s_4\}$ constitutes blocking in Program 2.3 because:

(a) the programmer has chosen a subgraph $\Delta' << \Delta$ such that no execution is intended to stay in Δ' forever (in this example Δ' is generated by the predicate $\{n_3 = 0\}$);

(b) T is a cycle in Δ';

(c) one may arrange that the time spent by each PE at any statement Pi is finite during execution of cycle T.

Note that (c) expresses a variant of the well-known finite delay property (FDP) and fairness property.

Cycle $T_1 = \{s_b \rightarrow s_b\}$ constitutes blocking in Program 2.4 for the similar reasons. In this case, the subgraph Δ' in (a) is generated by the predicate $\{n_7 = 1\}$ and the FDP in (c) is to be applied to PEs at all positions Pi, except for those PEs at position P7.

We wish to formulate (c) in a form generalizing both program examples above. To do this, we introduce the notion of an enabled position as follows:

Let T and Δ', $T << \Delta'$, be subgraphs in Δ. Position Pi is said to be enabled on T with respect to Δ', iff there exist a state s (i.e. a node) in T with $n_i > 0$ and an arc in Δ' starting at s whose corresponding transition is a move from Pi.

Examples: for Program 2.3 positions P1 and P2 are enabled on T with respect to Δ', position P3 is not; for Program 2.4 the only position enabled on T_1 with respect to Δ' is P5.

We now reformulate (c) as the following infinite loop property (ILP):

Cycle T in a subgraph Δ' of the reachability graph Δ satisfies the ILP with respect to Δ', iff each position Pi wich is enabled on T with respect to Δ' is also enabled on T with respect to T.

Now we discuss a possible way to test these properties algorithmically.

Note that the skeleton code includes no indication to a choice for Δ' in (a). For Program 2.3 subgraph Δ'' generated by the predicate $\{n_2 = 0\}$ contains cycle $T' = \{s_1 \rightarrow s_2 \rightarrow s_1\}$ which satisfies (b) and (c). This "blocking" by no means bothers the programmer. But from the text of Program. 2.3 it is not clear why Δ' was prefered over Δ''. Thus an indication of subgraph Δ' should be supplied by the programmer. The latter is supposed to have a semantical reason for this indication. If this indication is in a form of a computable predicate, then Δ' may be generated by known methods once Δ is given. Finding cycles as in (b) is also a standard computational problem.

We formulate a condition of non-fulfilment of the (ILP) for an irreducible cycle T.

The ILP fails for an irreducible cycle T if:
(non-ILP1) the length of T is greater than 1 and there is a position Pi such that n_i is a positive constant during execution of T,
or (non-ILP2) the length of T is 1 (i.e. it consists of a single state looping to itself) and there are at least two different positions Pi and Pj such that both n_i and n_j are positive constant during execution of T.

The above defined ILP expresses the FDP in Bose semantics. Indeed, let Θ be the canonical mapping from the Boltzmann representation to the Bose representation. ($\Theta(s') = s$, iff each state coordinate n_i of s represents number of PEs at Pi as indicated by s'.) Given a cycle $T = \{s_1, s_2, \ldots\}$ of the program execution in Bose semantics satisfying the ILP, one can arrange a cycle of the program

execution $T' = \{s'_1, s'_2, \ldots\}$ in Boltzmann semantics satisfying the FDP and such that $\Theta(s'_1, s'_2, \ldots) << (s_1, s_2, \ldots)$. (Note that several nodes in T' may correspond to a single node in T, since T' might make several convolutions over T in order to force each PE at each enabled position to move.)

2.12. Eliminating private variables.

Given a program with private variables which can take on only a fixed number of distinct values, we show how to define a new program without private variables equivalent to the first one.

We assume that our skeleton program has a fixed number ℓ of private variables $\pi = \{p_1, \ldots, p_\ell\}$. In addition to statements (I), (E) as in section 2.7 the program might have (interior) statements of the form

(I_π) Pi: go to $d_i(F\&f_i(\pi))$

where d_i and f_i are some directing and replacing functions as in (I). The execution state of such a program is the vector

(2.8) $s = (\pi^1, Pi^1, \ldots, \pi^N, Pi^N; c_1, \ldots, c_r)$,

where N is the total number of PEs, π^j is the current value of π and Pi^j is the current executable statement for PE_j, the counters c_i are as above. Note that the execution state is represented here in Boltzmann semantics.

Let z be the total number of distinct values of π, and let G be the graph of the program as in section 2.7. Then the node set of the graph G′ of the new program (without private variables) is

{the node set of G} × {set of possible values of π}.

The node set of G′ may be thought of as z copies of the node set of G. We call each such copy a slice. The slices are naturally enumerated by values of π, and a node of G′ (a position of the new program) may be written in the form Pi × π, where Pi is a node (a position) of the old program and π is a value of the vector of private variables. To each (I) or (E) statement Pi from the old representation there correspond z copies {Pi × π} in the new representation, one copy per slice. To a statement (I_π) in the old representation (when a PE replaces value π of its private variables by $f_i(\pi)$ and moves from Pi to $P(d_i(\pi))$ there correspond z statements of the form

(2.9) $\qquad$ Pi × π: go to $P(d_i(\pi))$ × $f_i(\pi)$.

where π runs all z possible values. Statement (2.9) means: jump from the node representing Pi in slice π to the node representing $P(d_i(\pi))$ in slice $f_i(\pi)$.

The arcs of G′ are: 1) inside each slice all one-to-one copies of arcs corresponding to (I) and (E) statements in the old representation; 2) arcs between slices corresponding to statements (3.3) defined as it was explaned above.

Note that reductions in the obtained skeleton program are possible.

As an example consider a busy-wait synchronization routine, suggested by Kalos and Lubachevsky [12]. This routine has application in the following situation:

Program 2.6

```
cycle: {working; executed for a finite time}
       synch
```

The purpose of the "synch"-routine is to trap PEs until all of them complete a previous asynchronous section and then to release them for execution of the next asynchronous section.

In Program 2.7 counters count(1) and count(2) are used to calculate the number of PEs trapped by the routine. They work in a flip-flop manner so that count(1) / count(2) is used by all odd / even invocations of the routine. The private variable index is 1 (2) for odd (even) invocations. (Note that obvious one-counter solution is incorrect.)

Program 2.7

```
COMMENT initially count(1) = count(2) = 0, index = 1.

     P1:  {working; executed for a finite time}

          if  F&A (count(index), 1) ≥ N-1  then go to P3

COMMENT all PEs but the last one are busy-waiting

     P2:  if  count(index) ≥ 1  then go to P2
          index <- 3-index                                  synch
          go to P1

COMMENT the last PE releases all PEs

     P3:  count(index) <- 0
          index <- 3-index
          go to P1
```

Note that Program 2.7 does not completely correspond to the language described in section 2.7. In particular, the use of variable index is not supported in this language. One may view Program 2.7 as an abbreviation for the (more lengthy) code, written according with the rules of section 2.7. For example the following fragment in the short code

```
Pi:    sem(index) <- 0

Pj:...
```

will be rewritten in the long code as follows:

```
Pi':  go to Pi_index

Pi_1:  sem(1) <- 0

         go to Pj'

Pi_2:  sem(2) <- 0

Pj':...
```

Here new labels Pi', Pj' correspond to the old labels Pi, Pj respectively.

The procedure of eliminating private variables suggests a duplication of the (long) code, since each private index takes on two values. Statement index <- 3-index of the old code is to be replaced by two copies in the new code, one copy per slice, each of which is a null statement followed by a "go to" to its brother in the opposite slice. There are several statements in each slice which will never be executed (for example, statement sem(2) <- 0 in the slice corresponding to index=1). If one eliminates these dead statements and merges null statements with their successors then the following skeleton program is obtained:

Program 2.8

```
P1:  if  F&A (count(1),1) ≥ N-1  then go to P3

P2:  if  count(1) ≥ 1  then go to P2
     go to P4

P3:  count(1) <- 0

P4:  if  F&A (count(2),1) ≥ N-1  then go to P6

P5:  if  count(2) ≥ 1  then go to P5
     go to P1

P6:  count(2) <- 0
     go to P1
```

2.13. Indivisibility by virtue of the program.

A classical example of indivisibility by virtue of the program is presented by the CS in Program 2.1, if property (CSe) is satisfied.

Any public memory reference executed within the CS is indivisible by virtue of this program providing that there is no reference to the same public memory location outside the CS. Another example is statements P3 and P6 in Program 2.8. It can be proven [12] that no more that 1 PE can execute P3 or P6 at a time and that statement P1 (P4) can not occur simultaneously with P3 (P6). Note that the Ultracomputer supports indivisibility of write's and F&As which concurrently reference a public variable. But even if a parallel computer does not support such indivisibility, and only combinations of concurrent F&As and (separately) read's and a single write are supported, the semantics of execution of Program 2.8 are still adequately represented by the F&f-model.

We now generalize these two examples. We say that a computer supports the total indivisibility (of shared memory references) for a given F&f-program if any combination of F&f-references to the same memory location is executed by the computer in a way equivalent to some serial execution of these references. Thus for any non-negative n_1, n_2, and n_3 the serialization of any combination $P1^{n_1}P2^{n_2}P3^{n_3}$ of executions of statements P1, P2, and P3 in Program 2.8 must be supported in the case of total indivisibility.

Let $p = p(s)$ be a predicate on a program state s as in (2.3) and let our computer only support indivisibility in those combinations of F&f-statements for which the state s satisfies the predicate p. Then we say that the computer supports the partial indivisibility (of shared

memory references) with respect to the predicate p. Such a predicate p for Program 2.8 might be $p = \{n_1 \times n_3 = n_4 \times n_6 = 0,\ n_3 \leq 1,\ n_6 \leq 1\}$.

Note that when trying to verify a program under the assumption of partial indivisibility, one should first check the consistency of the proposed F&f-model of the program, i.e. indivisibility of each F&f-statement. It is possible, for example, to encounter a reachable state s in which one PE is attempting to write a private value into a public location c and another PE is concurrently attempting to increment c by a private value, without indivisibility of concurrent write and increment being supported by the computer. In this case either one can not build the reachability graph, or the graph, built under the assumption of indivisibility of concurrent write and increment, will not cover all the possibilities of the real parallel execution. It is very easy to prove the following

Proposition. Let Δ be the reachability graph of a F&f-program built under the assumption of total indivisibility, and let p be a predicate valid for any reachable state in Δ. Then it is possible to build the reachability graph Δ' of this program under the assumption of partial indivisibility with respect to p and $\Delta' = \Delta$.

This simple proposition has one important application as a resolution rule for verifying critical sections. We will describe this rule in the example of Program 2.4. The computer which is supposed to execute Program 2.4 is responsible only for the indivisibility of P and V operations. Indivisibility of accesses to location g (in statements

P2, P3, P8, and P9) must be proven before attempting to establish any other property. One method of constructing such a proof might be to ignore the contents of these statements as was done with the CS in Program 2.2. in rewriting the code in the form of Program 2.3. This method does not work here since Program 2.4 contains test-branching of g (in statements P3 and P9). Assuming references to g to be indivisible a priori is seemingly circular reasoning (their indivisibility is what we are trying to prove). Nevertheless, according to the above Proposition we may temporarily accept this unproved assumption and then prove either it or its negation as in the following procedure:

Step 1. Build the reachability graph under the assumption of total indivisibility.

Step 2. Verify the predicate p = {no more than 1 PE may access the critical sections in question} in the reachability graph. (Note that in Program 2.4 the predicate p may also be expressed as: {no more than 1 PE may access g}.)

Step 3. If there is a state which violates property p then the required indivisibility is not obtained (and one should fix the bug before verifying any other property). Otherwise, the indivisibility is supported by virtue of the program itself.

3. Verifier-2.

The algorithm called verifier-2 helps to prove the correctness of parallel coordination programs for arbitrary number N of PEs. This algorithm is rather complex (a FORTRAN program which implements only a

subset of it contains about 4000 lines) and it will be described in part II of this paper. Nevertheless proofs produced by verifier-2 may be understood and checked without a knowledge of its workings. In particular, these proofs are valid even though verifier-2 itself may contain bugs. In this section an example of verification using verifier-2 is discussed and several basic notions concerning verifier-2 are introduced.

3.1. Correct semaphore.

The following code presents a correct solution to the problem stated in section 2.2.

Program 3.1

```
COMMENT sem is public; initially sem = 1;
        no {...} section accesses sem

     P1:  if  sem ≤ 0  then go to P1                    |
                                                        |
     P2:  if  F&A(sem,-1) ≥ 1  then go to P4            |
                                                        |   entry
     P3:  F&A(sem,1)                                    |
          go to P1                                      |

     P4:  {critical section; executed for a finite time}

          F&A(sem,1)                                    |   exit

          {remainder of cycle; executed for a finite time}
          go to P1
```

Observe by comparison with the "naive" Program 2.2 that this program contains an extra checking (statement P1).

Verifier-1 was applied to this program for various N to check properties (CSe) and (CSr) stated in section 2.2. All these trials acknowledged properties (CSe) and (CSr). This increased a psychological confidence in the program correctness but did not constitute a mathematical proof valid for any N.

3.2. Reachability set description.

Descr. 3.1 was produced by verifier-2. It is a compact description of the reachability graph Δ of Program 3.1 and it is valid for arbitrary N. We will call Descr. 3.1 a reachability set description (RSD) for the program.

Descr. 3.1

A $n_1 + n_2 = N$, sem = 1
moves from P1 to P2 lead to A
moves from P2 to P4 lead to B

B $n_1 + n_2 + n_3 + n_4 = N$, sem = $1 - n_3 - n_4$,
$n_4 \leq 1$,
$n_3 + n_4 \geq 1$,
$n_1 + n_4 \geq 1$
moves from P1 to P1 lead to B
moves from P2 to P3 lead to B
moves from P3 to P1 lead to B (if $n_4 = 1$ or $n_3 \geq 1$)
moves from P3 to P1 lead to A (if $n_3 = n_4 = 0$)
moves from P4 to P1 lead to B (if $n_3 \geq 1$)
moves from P4 to P1 lead to A (if $n_3 = 0$)

In Decr. 3.1 the reachability set $R(s_1)$ for the initial state $s_1 = \{n_1 = N, n_i = 0, i = 2,3,4; \text{sem} = 1\}$ is represented as the union of two metastates A and B. Each of them is a set of states described by a number of equalities or inequalities relating the coordinates of the

state-space vector $s = (n_1, n_2, n_3, n_4; sem)$. (We imply that $n_i \geq 0$, $i = 1, 2, 3, 4$ and that n_i and sem are integers.)

A = A(s) and B = B(s) may be thought of as predicates on a state of the program. For example, $A(s) = \{n_1 + n_2 = N\} \& \{sem = 1\}$. It will be proved that predicate [A(s) OR B(s)] is the strongest program invariant. As in sections 2.5, 2.10 the sizes of Δ in Bose and Boltzmann semantics may be computed given the expression above for [A **OR** B]: the size in Bose semantics is $N^2 + N + 1$, the size in Boltzmann semantics is $N \times 3^{N-1} + 3^N - 2^N + 1$.

The phrases listed under the formulas for the metastates are called directing phrases. A directing phrase represents a class of moves "from Pi to Pj". A particular move represented in a directing phrase can be identified with a pair (s_1, s_2) of states of the program, or else it can be identified with an arc of Δ.

Note that for given Pi and Pj and a given metastate, "moves from Pi to Pj" indicated for this metastate as the source (if any such moves exist) may direct to different destination metastates. For example two classes of "moves from P3 to P1" are indicated for metastate B in Descr. 3.1. Moves of the first class "lead" to B, moves of the second class "lead" to A. Branching conditions written in parentheses after the two directing phrases distinguish these two classes. Note that the predicate on the program states which expresses the branching condition is understood for states resulting from the move. For example, if $n_4 = 1$ or $n_3 \geq 1$ for the state s obtained after the move of a PE from

P3 to P1 and starting at a state $s' \ll B$, then $s \ll B$. Otherwise (i.e. if $n_3 = n_4 = 0$ for state s), $s \ll A$.

3.3. Proof of correctness of the semaphore program.

Proof of the fact that Descr. 3.1 correctly represents the graph Δ follows in section 3.5. We now assume that Descr. 3.5 is correct and derive from it a proof of properties (CSe), (CSr) of Program 3.1. Since Descr. 3.1 represents the graph Δ for arbitrary N, this proof will be valid for arbitrary N as well.

Property (CSe) is immediate from Descr. 3.1.

Let us show property (CSr). First, we derive from the RSD a compact description of the subgraph Δ' generated by the predicate $p = \{n_4 = 0\}$. The latter expresses the condition of "being out of the critical section" in the considered program. We will use the notation S&p for the intersection in the state-space of set S and the carrier of the predicate p. S&p may also be understood as the conjuction of two predicates: p and {being an element of S}.

Δ' is presented in Descr. 3.2.

Descr. 3.2

A&p $n_1 + n_2 = N$, sem = 1
moves from P1 to P2 lead to A&p

B&p $n_1 + n_2 + n_3 = N$, $n_1 \geq 1$, $n_3 \geq 1$, sem = $1-n_3$
moves from P1 to P1 lead to B&p
moves from P2 to P3 lead to B&p
moves from P3 to P1 lead to B&p (if $n_3 \geq 1$)
moves from P3 to P1 lead to A&p (if $n_3 = 0$)

Descr. 3.2 was obtained from Descr. 3.1 by following transformations:

1) form the conjunction of sets A and B with p;

2) since $n_4 = 0$, eliminate all directing phrases which contain references to P4;

3) form the conjunction of branching conditions with p.

Our next goal is to derive from Descr. 3.2 a compact description of all cycles in Δ'. To any set description like Descr. 3.1 or Descr. 3.2 we associate an auxiliary graph Γ as follows. The nodes of Γ are metastates and the arcs of Γ are associated with directing phrases. A directing phrase of the form "moves...lead to Y" attached to the description of metastate X corresponds to an arc X -> Y. We call graph Γ the associated graph. To Descr. 3.2 there corresponds a graph Γ with 2 nodes (A&p and B&p) and 4 arcs (A&p->A&p, B&p->B&p, B&p->B&p, B&p->B&p and B&p->A&p).

There are two cycles in Γ (A&p->A&p and B&p->B&p). Below we represent fragments of Descr. 3.2 to which each of these cycles corresponds.

The first cycle in Γ corresponds to the following fragment:

Descr. 3.3

A&p $n_1 + n_2 = N$, sem = 1
moves from P1 to P2 lead to A&p

The second cycle in Γ corresponds to the following fragment:

Descr. 3.4

B&p $n_1 + n_2 + n_3 = N$, $n_1 \geq 1$, $n_3 \geq 1$, sem = $1-n_3$
moves from P1 to P1 lead to B&p
moves from P2 to P3 lead to B&p
moves from P3 to P1 lead to B&p, if $n_3 \geq 1$

Observe that there exists a standard mapping from the original graph Δ' (with which Descr. 3.2 is associated) to the associated graph Γ.

In an obvious sense this mapping agrees with the representation of Δ' by Descr. 3.2. To each node (arc) of Δ' there corresponds a node (an arc) of Γ, which is itself associated with the metastate (directing phrase) that represents this node (arc) of Δ'. Each a cycle in Δ' must correspond to a cycle in Γ. Therefore a description of a cycle in Δ' (if any such cycle exists) must be contained completely in one of the two fragments above. In other words the states of this cycle may not jump from one fragment to the other.

Descr. 3.3 does not describe any cycle in Δ'. (Since graph P1->P2 contains no cycle.)

Descr. 3.4 contains a class of cycles in Δ' presented in Descr. 3.5.

Descr. 3.5

B&p $n_1 + n_2 + n_3 = N$, $n_1 \geq 1$, $n_3 \geq 1$, sem = $1-n_3$
moves from P1 to P1 lead to B&p

Descr. 3.4 does not describe any cycle different from those presented in Descr. 3.5 for the following reason. Consider the graph $G' = \{P2 \rightarrow P3 \rightarrow P1 \rightarrow P1\}$. G' represents a fragment of the skeleton program such that each PE which executes along a cycle in Δ' can not leave G'. The only possibility for such a PE is that it moves along some cycle in G'. But $\{P1 \rightarrow P1\}$ is the only cycle in G'. Thus Descr. 3.5 presents the only type of cycles in Δ'.

Now to prove (CSr) one should check these cycles with respect to the ILP. Observe that each individual cycle in Δ' presented in Descr. 3.5 is irreducible and that condition (non-ILP1) from section 2.11 is met with the two positions mentioned in (non-ILP1) being P1 and P3.

Since ILP can not be satisfied, property (CSr) is fulfilled.

3.4. Comparison with the method of verification by Owicki and Gries.

A referee mentioned that it is easy to prove all properties of the considered programs using the method of Owicki and Gries[13, 14]. This section is included in order to compare their method with ours in the verification of a parallel program.

First, we give here an outline of a proof of (CSe) using the method of [13, 14]. Program 3.1 possesses an invariant $q = \{sem = 1 - n_3 - n_4\}$. (Note that this q may be generated mechanically using the generalization [2] of the semaphore invariant method [8].) Property (CSe) may be expressed as the predicate $r = \{n_4 \leq 1\}$ and it may be derived from existence of q by observing that q & r is also an invariant. Finally, q & r implies r which is (CSe). This concludes the proof.

But this method does not explain how to find a proper q for a given r in a general case. The invariant q which appeared helpful in this particular case may not work in other cases.

For example, suppose one tries to prove the property "it is not possible for all the PEs to simultaneously execute statement P3". This property may be expressed by the predicate $r_1 = \{n_3 \neq N\}$. Now the invariant q does not work, since $q \,\&\, r_1$ is not an invariant. The strongest program invariant q_* is the ultimate solution. Namely, were $q_* \,\&\, r_1$ not an invariant, the required q would not have existed and

$r_1(s)$ would have been false for some reachable state s. Verifier-2 supplies RSD and $q_*(s)$ = [A(s) OR B(s)]. Since [A(s) OR B(s)] & $r_1(s)$ is an invariant, property r_1 is proved true.

One may object to the latter example in that property r_1 presents no semantic interest in Program 3.1. As a counter-objection we suggest one consider the program in [2] p. 344. It is not possible to prove one semantically meaningful property of the considered form for this program, if one only uses the invariant produced by [8].

Proving properties like (CSe) or r_1 is just a minor task the RSD helps to perform. We now compare the two methods in verifying (CSr), when the RSD works in "full capacity".

Note that Program 3.1 is not supposed to terminate, and the method [13, 14] is not formally applicable to this program. We can easily rewrite Program 3.1 in a "terminating" form by eliminating the "go to P1" in statement P4 and adding a "no-operation" statement P5 at the end of the program. Program 3.2 is the skeleton of the resulting code.

Program 3.2

```
COMMENT sem is public; initially sem = 1;

     P1:  if  sem ≤ 0  then go to P1

     P2:  if  F&A(sem,-1) ≥ 1  then go to P4

     P3:  F&A(sem,1)
          go to P1

     P4:  F&A(sem,1)

     P5:
```

Our method still works for this modified program. For the set of initial states $S1 = \{n_1 + n_5 = N\}$ verifier-2, produces the RSD as in Descr. 3.6.

Descr. 3.6

A $n_1 + n_2 + n_5 = N$, sem = 1
moves from P1 to P2 lead to A
moves from P2 to P4 lead to B

B $n_1 + n_2 + n_3 + n_4 + n_5 = N$, sem = $1 - n_3 - n_4$,
$n_4 \leq 1$,
$n_4 + n_5 \geq 1$,
$n_3 + n_4 \geq 1$
moves from P1 to P1 lead to B
moves from P2 to P3 lead to B
moves from P3 to P1 lead to B (if $n_4 = 1$ or $n_3 \geq 1$)
moves from P3 to P1 lead to A (if $n_3 = n_4 = 0$)
moves from P4 to P5 lead to B (if $n_3 \geq 1$)
moves from P4 to P5 lead to A (if $n_3 = 0$)

Using this RSD it is easy to prove the termination for Program 3.2 in the way similar to our proof of absence of livelock for Program 3.1.

The sufficient condition of termination of [13] is formally applicable to the Program 3.2. It states that this program terminates if each process "can be proved to terminate under the assumption that it does not become blocked." To prove termination of each individual process under this assumption, [14] suggests constructing an integer non-negative function t, such that executions decrease value of t. (It is not explained how to construct such a function.) Note that the non-blocking assumption is always fulfilled in our examples. In fact, a F&f-program has neither AWAIT nor similar constructs which would allow a PE to await a condition. Each PE always executes.

3.5. Reachability tree and forest.

The assertion that Descr. 3.1 represents the reachability graph Δ for any N must be proved. Such a proof consists of establishing the validity of the following two statements: (i) any reachable state lies in a set of the RSD (either in A or in B); (ii) any vector presented in the RSD (i.e. that from A or from B) is a reachable state.

Property (i) will be proved if we establish that [A(s) OR B(s)] is an invariant with the initial state ($n_1 = N$, $n_2 = n_3 = n_4 = 0$, sem = 1).

Descr. 3.1 itself contains enough information to make a proof of (i) just a mechanical task. In fact, it is immediate that the initial state belongs to A. Since the RSD contains description of all possible moves and all moves are to sets given in the RSD, (i) may be checked on

the case-by-case basis given the RSD. In this sense we say that the RSD is a proof of (i).

Similarly, we say that the proof of (ii) is Descr. 3.7, representing the so-called reachability tree for Program 3.1.

Descr. 3.7

S1 $n_1 = N$, sem = 1
moves from P1 to P2 lead to S2

S2 $n_1 + n_2 = N$, sem = 1
moves from P2 to P4 lead to S3

S3 $n_1 + n_2 + n_4 = N$, sem = 0
$n_4 = 1$
moves from P2 to P3 lead to S4

S4 $n_1 + n_2 + n_3 + n_4 = N$, sem = $1 - n_3 - n_4$
$n_4 = 1$
moves from P4 to P1 lead to S5

S5 $n_1 + n_2 + n_3 + n_4 = N$, sem = $1 - n_3 - n_4$,
$n_4 \leq 1$,
$n_3 + n_4 \geq 1$,
$n_1 + n_4 \geq 1$

This description was mechanically produced by verifier-2. It has the following meaning. Like the RSD above it lists some metastates (here S1, S2, S3, S4, and S5) and some moves. However Descr. 3.7 has a different overall structure.

The metastates Si can be viewed as the nodes of a directed tree (sequence in this example). S1 is the root and consists of a single element which is the initial state of the program. Each new Si down the tree adds up some new states and any set Si can be reached from its ancestors in the tree.

For example (we consider the hardest case), let us prove that any state s of S5 either lies in one of the metastates Si, i = 1, 2, 3, 4, or can be obtained as a result of a move of one or more PEs from P4 to P1 at a state that lies in S4.

Let the state $s = (n_1, n_2, n_3, n_4; sem)$ be an element of S5. According to the description of S5 there are two possibilities: either $n_4 = 0$ or $n_4 = 1$. If $n_4 = 0$, then $n_1 \geq 1$ and state $s = (n_1, n_2, n_3, 0; 1-n_3)$ can be obtained by a move of a PE from P4 to P1 at state $(n_1-1, n_2, n_3, 1; -n_3)$ which lies in S4. If $n_4 = 1$, then state $s = (n_1, n_2, n_3, 1; -n_3)$ belongs to S4.

Now we show (ii). First we see that any state s from the union $R \overset{Df}{==} S1 \cup S2 \cup S3 \cup S4 \cup S5$ is reachable. Second we observe that $R << A \cup B$ (in fact, one has $S1 << S2 = A$, $S3 << S4 << S5 = B$). Hence (ii) is proved.

In the example of Program 3.1 the metastate S1 reduces to a single initial state. In the example of Program 3.2, the metastate S1 contains more than one state. Generally we allow S1 to be expressed in a disjunctive normal form of atomic predicates each of which is a linear inequality of the form

$$(3.1) \qquad s_1 n_1 + \ldots + s_k n_k \quad \{ \underline{\leq} \} \quad b$$

on state coordinates. We discuss restrictions on coefficients of (3.1) in section 3.10.

In such a general cases one might have a forest instead of a tree as above. Each conjunction in S1 will then correspond to a root of a tree of this forest. This forest is called a reachability forest.

3.6. A geometric interpretation of the reachability tree and the RSD.

For a given N consider the 3-dimensional grid H formed in the 4-dimensional space of vectors (n_1,n_2,n_3,n_4) by all non-negative integer solutions of the equation

$$n_1 + n_2 + n_3 + n_4 = N.$$

Since Program 3.1 possesses the invariant q = {sem = 1 - n_3 - n_4}, a state (in Bose semantics) of this program may be uniquely identified by a point of H. (It is difficult to draw the graph of this 3-dimensional construction. A simpler example in which H is 2-dimensional will be presented in section 3.8.)

We introduce the bundle of possible transitions (or simply the bundle) corresponding to each s << H. This bundle will be denoted as C(s). C(s) is a set of transitions. A transition of C(s) may be thought of as a 4-dimensional integer vector t such that vector s+t is a state resulting from a PE completing execution of a statement, if the starting state was s. We denote by t_{ij} the transition vector corresponding to a PE moving from Pi to Pj. Hence t_{ii} = 0 and if i ≠ j, t_{ij} is a vector all of whose components are zero save the i-th which is -1 and the j-th which is 1.

We now investigate the reasons why the bundles C(s) of different states s can differ. It follows that if there is no PE at Pi in state s (the i-th coordinate of s is 0), then C(s) contains no t_{ij} for any j. This is one reason why C(s′) can differ from C(s"). Were this the only reason, then our geometrical construction might be qualified as a vector addition system (VAS).

In a VAS a state freely migrates from a grid point s′ to a grid point s" providing vector s"-s′ is a multiple to a transition from a bundle. The only obstacle which may prevent such moving is an "outside boundary" which represents the condition of non-negativity of a coordinate of the state vector.

There exists another reason why C(s′) can differ from C(s") in the system we describe. This reason may be interpreted geometrically as an "inside boundary". This "boundary" may be thought of as a plane L with the equation $n_3 + n_4 = 1/2$. L divides H into two regions, region H1, inside which $n_3 + n_4 \leq 0$, and region H2, inside which $n_3 + n_4 \geq 1$.

If s $\ll$ H1 then sem ≥ 1. This means that no PE can move from P2 to P3, or else C(s) cannot contain vector t_{23} (but may contain vector t_{24}).

If s $\ll$ H2 then sem ≤ 0. This means that no PE can move from P2 to P4, or else C(s) cannot contain vector t_{24} (but may contain vector t_{23}).

The structure thus obtained is an example of a controlled vector addition system (CVAS). The notion of a CVAS will be formally introduced and studied in part 2 of this paper. A state may move in the state-space H of the CVAS along a given direction defined by a transition until an "obstacle" is met.

If the obstacle is an "outside boundary" then further moves in this direction are impossible. If the obstacle is an "inside boundary" then one more transition is effected in the given direction. If the bundle of the current point of the new region does not include this transition then further moves in this direction are impossible.

Now we can give the following geometrical interpretation of the process of producing Descr. 3.7 by verifier-2.

Metastate S1 reduces to a single state s_1. $C(s_1)$ contains the only transition t_{12}. The state moves along t_{12} as far as possible and thus creates metastate S2.

Bundles of the states of S2 contain at most two transitions: t_{12} and t_{24}. Verifier-2 checks transition t_{12} although it says nothing about this transition, since it generates no new state. When verifier-2 tries to apply transition t_{24} to the states of S2, the moving states immediately cross L and since bundles in H2 do not include t_{24} the moving stops. As a result, metastate S3 is generated in H2.

Bundles of the states of S3 contain t_{11}, t_{23}, t_{41}. Verifier-2 first tries t_{11} which does not produce anything new. Then it tries to apply transition t_{23} to states of S3, and the moving states produce S4 without crossing L. Verifier-2 realizes that S3 << S4. Hence it does not try the remaining t_{41} for metastate S3 but instead starts working with S4.

Bundles of the states of S4 contain t_{11}, t_{23}, t_{31}, t_{41}. Verifier-2 tries t_{11}, t_{23}, t_{31} without producing anything new. When it tries to apply transition t_{41} to the states of S4, the moving states produce S5 without crossing L.

Working with all possible transitions at S5, verifier-2 realizes that no more states can be produced. This terminates the phase of building the reachability tree.

Once the reachability tree is produced, verifier-2 reduces the set of all metastates as follows. It scans each metastate Si and discards it if Si is a subset of the union of other metastates remaining at the time of this scanning. (This is an easier way to think about this reduction. In the existing program, however, this reduction interleaves the process of building the reachability tree.)

Two metastates, S2 and S5, remain after the reduction. Verifier-2 completes the obtained description with all possible directing phrases by trying all possible transitions for both S2 and S5 and thus produces the RSD.

3.7. Compactness property.

A reachability tree or forest is the initial structure used to produce the RSD. Considering the problem of building the reachability forest, we note that the finiteness of such a forest infers a special compactness property for the program. We discuss this property in the context of the reachability tree of Descr. 3.7.

Consider any metastate S(i+1), $i \geq 1$, on the reachability tree in Descr. 3.7. Any state $s \ll S(i+1)$ is either a state of some Si' for $i' \leq i$ or it may be obtained from a state $s' \ll Si$ by several PEs moving simultaneously from one position to another according to the directing phrase attached to the description of Si. In the latter case, $s \lessdot F(s')$.

In other words, $S(i+1) \ll S1 \cup \ldots \cup Si \cup F(Si) \overset{Df}{==} Ki$. Since $S \ll F(S)$ for any S, one has $Ki \ll F(S1) \cup \ldots \cup F(Si)$ and $S1 \cup \ldots \cup S(i+1) \ll F^i(S1)$. Since $S1 \cup \ldots \cup S5 = R(S1)$ (this was proved in section 3.4), one has

(3.2) $$R(S1) \ll F^4(S1).$$

To express (3.2) in words: any reachable state of Program 3.1 may be reached from the initial state in 4 executional steps. Here "executional step" is a synonym for "application of the progress functional". More detailed analysis of this example shows that 4 in

(3.2) may be decreased to 3 since S4 U S3 << F(S2), i.e. any reachable state may be reached in 3 steps.

Remember from section 2.9 that any state s of a parallel program may be reached from the initial state s_1 in a number of executional steps bounded by a number independent of s if and only if the reachability graph is finite. Therefore it follows from (3.2) that the reachability graph Δ for our program is finite for any number N of executing PEs. Moreover, since (3.2) is true for any N, any state of the program may be reached from the initial state in at most 4 executional steps and this does not depend on how many PEs execute.

We can state this property in another way. Suppose that each statement Pi is effected in a single cycle by the parallel computer. At every cycle each PE either delays or executes its current statement. It follows that there is a time T_0 independent of N such that any state of the semaphore program can be reached from S1 within time T_0.

This property is inherent to the program and does not depend on the method by which we study the program. We call it compactness of a program (with respect to the set S1).

Compactness may also be understood as finiteness of the expansion (1.1) or as finite convergence of the sequence $F^i(S1)$ in the limit $i \to \infty$, if both these properties of finiteness take place uniformly with respect to N.

3.8. An example of a non-compact F&A-program.

Program 2.5 is not compact for the obvious reason that it has an unbounded counter c. For any fixed number of PEs, N, reachability graph Δ of this program is infinite. Below a less trivial example is presented.

Consider Program 2.3 and take sem = N-1 as the initial value for sem (instead of sem = 1). The example obtained possesses a finite Δ for any fixed N but is non-compact.

Fig. 3.1 presents the reachability graph Δ for this program for N = 4. Grid H in this example is a planar triangle and contains all (non-negative integer) solutions of the equation $n_1 + n_2 + n_3 = N$. In this case, the invariant {sem = 1 - n_2 - n_3} allows one to exclude sem from the description of a state. As is seen from Fig. 3.1 the graph Δ consists of all points of H except for the unreachable point (0,0,N). The "inside boundary" L indicated in Fig. 3.1 with stars divides H into two regions, H1 (the points below L) and H2 (the points above L).

The bundles are shown for points of each region. Note that those points which lie on the "outside boundary" might not have some of the transitions from the corresponding bundle. For example, the transitions t_{21} and t_{31} are not included in C(N,0,0).

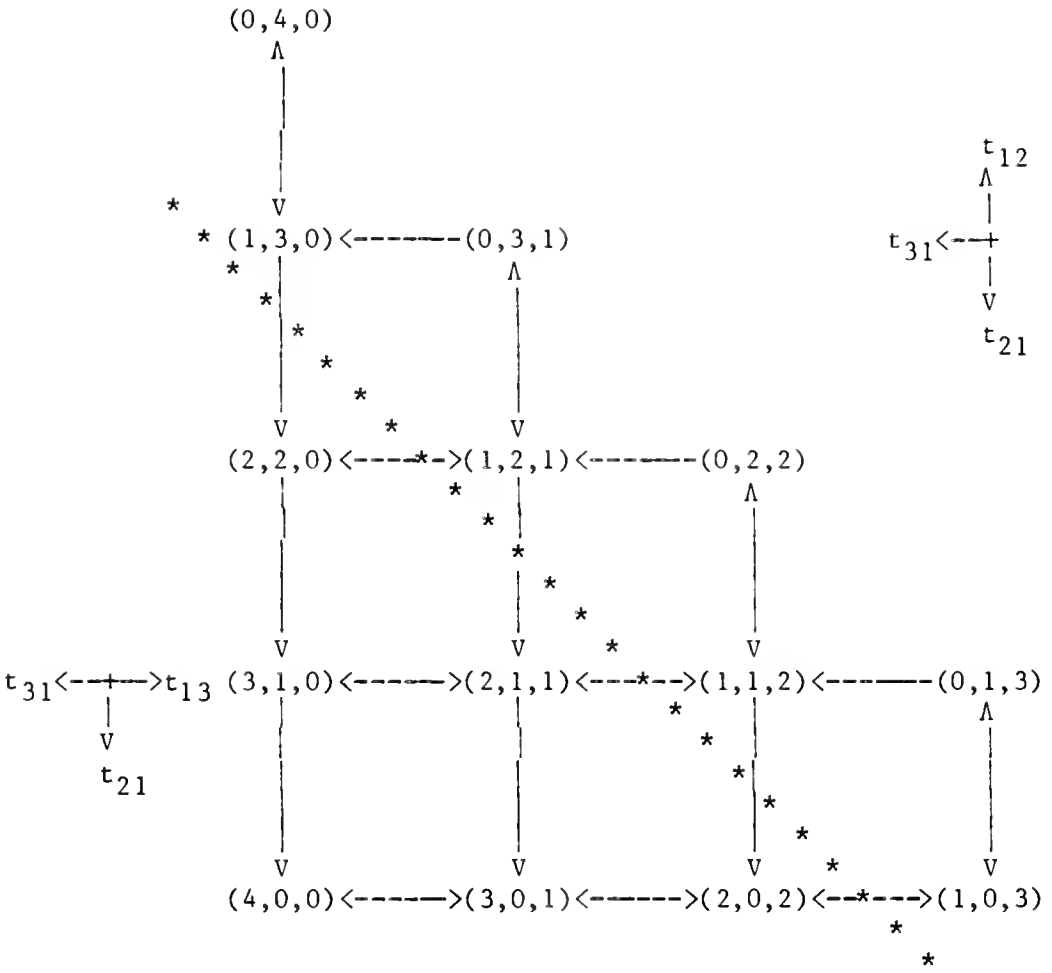


Fig. 3.1. The reachability graph for a non-compact program. (Program 2.2 with the initial value for sem = N-1. Here N = 4.)

Let $s_j \overset{Df}{==} \{0,j,N-j\}$. Observe from Fig. 3.1 that s_{j+1} not$\ll$ $F(s_j)$ and that any path from s_1 to s_N contains $s_2,\ldots s_{N-1}$. Therefore s_N not$\ll$ $F^{N-1}(s_1)$. It is clear from Fig. 3.1 that every path from (N,0,0) to s_N contains s_1. Therefore s_N not$\ll$ $F^{N-1}(N,0,0)$.

In other words, there exists a reachable state s_N that cannot be reached within N-1 steps. This implies the non-compactness.

A more detailed analysis of this example shows that s_N not$\ll$ $F^{2N-2}(N,0,0)$, but that the reachability set R(N,0,0) is exhausted by $F^{2N-1}(N,0,0)$. Note that if the initial state is (0,N,0), then this example becomes compact.

3.9. Neither a PV-skeleton nor a Petri Net can simulate a general F&A-skeleton.

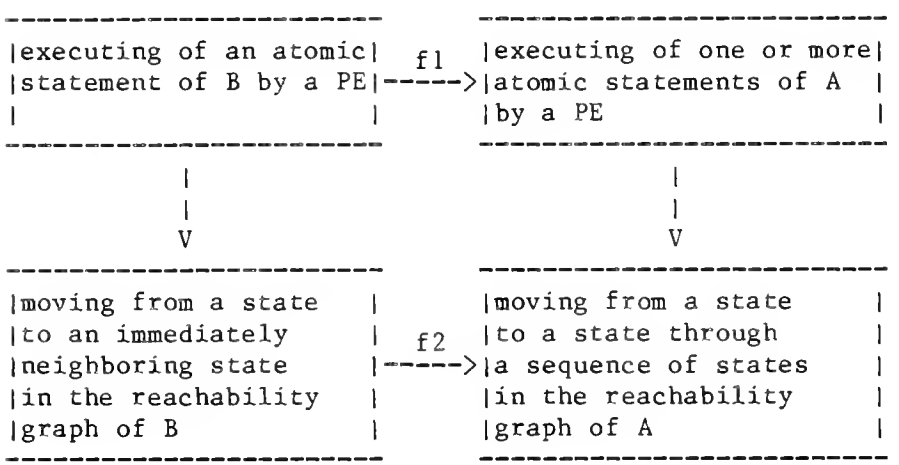
We first describe the meaning of "program A simulates program B". Since the detailed formal definition of that appears too large, we will substitute it with a somewhat intuitive definition (hopefully the reader could complete it with the necessary details).

Consider two F&f-programs, A and B. We say that program A simulates program B if there are two correspondences f1 and f2, such that f1 maps the code of B into the code of A and f2 maps the

reachability graph for B into the reachability graph for A, subject to two following conditions:

1) if s is an initial state of B then f2(s) is an initial state of A;

2) let P be an atomic statement of B. To each indivisible execution of statement P by a PE there corresponds an indivisible sequence of executions f1(P) of one or more atomic statements of A by a PE so that the following diagram is commutative:



In the above diagram the arrows pointing down represent the standard mapping from moving a PE between statements in the program code to the corresponding transition between states in the reachability graph.

If A simulates B, then in general A is more complex than B. And we will also indicate the fact that A simulates B by saying that B approximates A.

Note that if A simulates B and A is compact then B is compact.

We now wish to compare the expressive power of a F&A-program and a PV-program. We remind the reader that in a F&A-program (PV-program) the only indivisible operations on public variables supported by the computer are read, write, and F&A (P and V). Clearly any F&A-program may be simulated by a PV-program. Indeed, each occurrence of p<-F&A(c,i) may be replaced by the string

P(sem); p<-c; c<-c+i; V(sem),

where sem is a binary semaphore and p is a private variable.

We will therefore compare not the programs themselves but their skeletons, which are obtained by eliminating all operations except basic synchronization primitives.

Thus, a F&A-skeleton (a PV-skeleton) is a F&f-program in which only F&A-operations (P and V operations) are allowed on public variables (neither read nor write is allowed). Here we imply general P and V operations defined by statements (2.2p) and (2.2v), respectively, with the function f (m,h) = max (m + h, 0).

It is known that a PV-skeleton thus defined is equivalent to a Petri Net [10] and to a (restricted) VAS [7]. It will be proved in part II of the paper that a PV-skeleton is always compact. Therefore none of a PV-skeleton, a Petri Net, or a VAS may simulate a general F&A-skeleton. For example, the non-compact F&A-program described in section 3.9 can not be simulated in any of these ways.

3.10. Classes of parallel programs to which verifier-2 may be applied.

It is difficult to give a formal definition of a class of programs to which the proposed method of verification is applicable. Once such a definition is fixed, programs beyond the scope of the definition for which the method still works may emerge.

We can not, however, avoid giving a definition (though possibly too restrictive) for a class of parallel programs to which verifiers apply, since the verifiers are computer programs and their inputs must be specified precisely.

The input of verifier-1 is specified by the definition of a F&f-program in section 2.7 and by properties (v1), (v2), (v3) stated in section 1. The class of programs to which verifier-2 may be applied is more restrictive. In fact, we describe several classes of parallel program. As more is required from the class of programs, more is to be expected from the verifier. In the rest of this section, k denotes the total number of statements Pi in a F&f-program in question.

A F&f-program, as defined in section 2.7, is called a counter-conservative programs (a CC program) if, for each counter c_i, there exists a conservative function $\gamma_i(n_1,\ldots,n_k)$ such that for any reachable state (2.3) of the program one has

(3.3) $$c_i = \gamma_i(n_1,\ldots,n_k)$$

For example, Program 3.1 is a CC program since the invariance of $p(s) = \{sem = 1 - n_3 - n_4\}$ implies that the relation

(3.4) $$sem = 1 - n_3 - n_4$$

holds for any reachable state of this program. Note that relation (3.3) needs not be an invariant in order to hold for any reachable state of the program.

The state-space of a CC program can be simplified by eliminating the counters, and such a program can be rewritten in the variables n_i. For example, the skeleton of Program 3.1 can be rewritten as presented below.

Program 3.3

P1: with n_1,n_2,n_3,n_4 when $n_3 + n_4 \leq 0$ do
$n_1 \leftarrow n_1 - 1$, $n_2 \leftarrow n_2 + 1$

P2: with n_1,n_2,n_3,n_4 when $n_3 + n_4 \leq 0$ do
$n_2 \leftarrow n_2 - 1$, $n_4 \leftarrow n_4 + 1$

P3: with n_1,n_2,n_3,n_4 when true do
$n_3 \leftarrow n_3 - 1$, $n_1 \leftarrow n_1 + 1$

P4: with n_1,n_2,n_3,n_4 when true do
$n_4 \leftarrow n_4 - 1$, $n_1 \leftarrow n_1 + 1$

Program 3.3 may be thought of as the result of collapsing N copies of Program 3.1, one copy corresponding to one PE. We call Program 3.3. a collapsed (presentation of the original) program. The predicates in statements P1 and P2 in Program 3.3 are obtained from the corresponding

predicates in Program 3.1 by replacing occurrences of sem with the expression (3.4).

The semantics of Program 3.3 are as follows. A (serial) computer picks any one of the statements P1-P4 and tries to execute it over the state vector $s = (n_1, n_2, n_3, n_4)$. If the vector of the so obtained state, $s' = (n_1', n_2', n_3', n_4')$, satisfies the conditions $n_i' \geq 0$, $i = 1,\ldots,4$, then the state s is set equal to s'. Otherwise the state does not change. The four statements in Program 3.3 do not imply any order. Any of them may be chosen to be executed at any time.

The collapsed code consisting of statements of the form

(3.5) with s when COND(s) do OPER(s)

may be written for any CC program. We introduce the following restrictions on COND(s) and OPER(s) in each statement of the collapsed code:

(n1) COND(s) is a truth combination of atomic linear predicates of the form (3.1), where s_i are 0 or 1, and b is integer.

(n2) OPER(s) is an addition of a transition vector to the state.

A collapsed program whose statements satisfy (n1) and (n2) is a particular case of a linear simple concurrent language program (linear SCL program) considered in [2].

Note that unlike [2] where no parameter dependence was assumed, we consider a linear SCL program depending on a parameter. In our case this circumstance requires special consideration.

In general, OPER(s) and COND(s) may depend on N. For example, this may be the case if the conservative functions in (3.3) depend on N. The set of initial states of the collapsed program also depends on N.

We call the original F&f-program normal if it is a CC program (and hence allows collapsing as explained above) and if the following properties hold for statements of the form (3.5) of the collapsed code:

Properties (n1) and (n2);

(n3) Neither OPER(s) nor COND(s) depend on the number N of PEs;

(n4) The set of initial states S1 is the carrier of the conjunction of the predicate

(3.6) $$n_1 + \ldots + n_k = N,$$

with a predicate q(s) which has the same form as COND(s) in (n1) and which (like COND(s)) does not depend on N.

Note that COND(s) does not depend on N provided that the conservative functions in (3.3) do not depend on N. But we do not impose this requirement in our definition as it might prove too restrictive in some cases: conservative functions for the "readers and writers" programs described in sections 4.1 and 4.2 depend on N, but these programs are normal.

Thus verifier-2 may start to produce a RSD for any normal program. It will be proved in part II of this paper that if the normal program is compact then verifier-2 will eventually terminate and produce a RSD provided that sufficient memory is available to the verifier.

Verifier-2 will sometimes produce a RSD for a non-normal program. We will describe a class of programs broader than normal which may still allow verifier-2 to build a RSD.

We wish to introduce a class of programs close to CC programs, that we will call <u>almost CC</u> (ACC) programs. To do this, we first mention that, for a quite general class of programs that are not CC programs, one may express counters in a more general form than equiation (3.3), namely

$$c_i = \gamma_i(q; n_1,\ldots,n_k), \tag{3.7}$$

where q is an element of some abstract set of <u>auxiliary variables</u> Q, introduced to verify such a program. During execution of the program, these auxiliary variables may change along with the program variables c_i. Each statement Pi of the program may be augmented by an assignment of the form

$$q \leftarrow \theta_i\ (q; c_1,\ldots,c_r). \tag{3.8}$$

We treat a so augmented Pi by executing the statement (3.8) alongside and indivisible from the old containment of Pi. We should stress that

neither the variables q nor the assignment (3.8) exists in the real program residing in the computer. They are imaginary and are introduced to analyze the behavior of the program.[7]

By eliminating the c_i in equ. (3.8) in favor of the n_i as given in equ. (3.7), one is led to the form

$$q \leftarrow \phi\ (q;\ n_1,\ldots,n_k). \tag{3.9}$$

If one imposes no restriction on Q, its constructing may be done trivially: take as Q the set of histories of the values of all the counters. Such "auxiliary" variables do not simplify the analysis.

A F&f-program, as defined in section 2.7, is called an ACC program, if q in equ. (3.7) is an element of a finite set Q.

We leave the construction of the set Q and the functions θ_i in equ. (3.7) to the discretion of the programmer. Note, however, that in the particular case when the set of program counters may be split into two subsets so that for each counter c of the first subset there exists a conservation function (3.3) and the counters of the second subset take only a fixed number of values (like binary semaphores), q may be

[7] Augmenting the state by a number of auxiliary variables is a quite usual method for studying properties of dynamical systems, of which parallel programs are a special case.

defined to include all the counters of the second set. In order to make the RSD simpler, one should try to make dimension of q smaller.

We now mention that N copies of an ACC program may be collapsed into a single program as is done for a CC program. The state of the collapsed program is $s = (n_1,\ldots,n_k,q)$. The first k coordinates of this s will be refered to as the n-component, and the last coordinate as the q-component.

We introduce the properties (an1)-(an4) analogous to (n1)-(n4), with the following differences:

the coefficients s_i and b of the atomic formulas (3.1), which constitute PRED(s), may depend on the q-component (As in (n1) and (n3) these formulas are linear inequalities on the n-component and do not depend on N.)

OPER(s) may include a statement of the form (3.7) (As in (n2) and (n3) OPER(s) includes addition of a transition vector to the n-component and does not depend of N.)

The set Q does not depend of N.

If the collapsed program satisfies the properties (an1)-(an4), then the original program is called an <u>almost-normal</u> program. Verifier-2 also can be applied to an almost-normal program.[8] A

[8]As the author knows no example of a compact almost-normal program for which verifier-2 does not terminate, it may be a reasonable conjecture that compactness is a sufficient condition for the termination of the verifier-2 when applied to an almost-normal program; however, no proof is known to the author.

metastate in the RSD for such a program is characterized by the set of inequalities (3.1), as before, and additionally by a value of q.

4. Examples of programs proved correct using verifier-2.

4.1. Readers and writers.

The following program was suggested in [5] as a solution for the readers/writers problem [3]. The code below is written in the "goto"-style with labels Pi inserted according the rules of section 2.4.

Program 4.1

```
COMMENT sem is public; initially sem = N;
        no {...} section accesses sem

    P1: {exterior code; executed for a finite time}
        go to P2 (to read) or to P6 (to write)

    P2: if  sem ≤ 0  then go to P2                    |
                                                      |
    P3: if  F&A(sem,-1) ≥ 1  then go to P5            |
                                                      |   read entry
    P4: F&A(sem,1)                                    |
        go to P2                                      |

    P5: {critical read section; executed for a finite time}

        F&A(sem,1)                                    |   read exit
        go to P1

    P6: if  sem ≤ N-1  then go to P6                  |
                                                      |
    P7: if  F&A(sem,-N) ≥ N  then go to P9            |
                                                      |   write entry
    P8: F&A(sem,N)                                    |
        go to P6                                      |

    P9: {critical write section; executed for a finite time}

        F&A(sem,N)                                    |   write exit
        go to P1
```

Note that the statement P1 in Program 4.1 is an example of an exterior position (cf. section 2.7). Also note that both the read and write sections of this code begin with extra checking statements, P2 and P6, respectively. Their inclusion in the code is inspired by the necessity of the similar extra checking statement in Program 3.1.

The following properties were proved correct using verifier-2:

(RWe) No more than 1 writer can write (i.e. stay in P9); while the writer is writing no reader can read (i.e. stay in P5).

(RWRr) Suppose no PE ever attempts to write. For any $k < N$, if k PEs are in the reader CS (i.e. staying at P5), then any number up to N-k PEs which start executing their reader entry section will eventually enter the reader CS, and the delay of these PEs will not depend on how long those k reader PEs remain in the CS.

(RWWr) Suppose no PE ever attempts to read. For any time t there exists a time $t' \geq t$ such that a PE is in the write CS at time t'.

(RWr) For any time t there exists a time $t' \geq t$ such that a PE is in one of the two CSs (i.e. at P5 or at P9) at time t'.

(RWe) stands for Reader-Writer-exclusion and is simular to property (CSe) in section 2.2. (RWRr) and (RWWr) stand, respectively, for Reader-Writer-Reader-reachability and -Writer-reachability. (RWr) stands for Reader-Writer-reachability. (RWRr), (RWWr), and (RWr) are simular to (CSr) in section 2.2.

All these properties were proved using the RSD produced by verifier-2. Note that (RWr) reduces to (RWWr) if there are no readers.

An alternative way of proving (RWe) might be to take the invariant $p_{rw} = \{sem = N - n_4 - n_5 - N \times (n_8 + n_9)\}$ and prove that $\{n_5 \times n_9 = 0\}$ & p_{rw} is an invariant (cf. proof of (CSe) by Owicki and Gries method in section 3.3). Property (RWRr) can be proved quite easily without using the RSD. The author does not, however, know of any proof of (RWr) except for that using RSD. This proof is given in [12].

Note that Program 4.1 is normal. Although the conservative function supplied by the invariant p_{rw} depends on N, the conditions on sem in statements P2, P3, P6, and P7 may be rewritten in the collapsed code without N. For example, $sem \geq N$ is the same as $n_4 + n_5 + n_8 + n_9 \leq 0$. The predicate $p_1 = \{sem \leq 0\}$ is slightly weaker than $p_2 = \{n_8 + n_9 \geq 1\}$: a state s with $n_4 + n_5 = N$ satisfies p_1 not p_2. Nevertheless p_1 can be replaced by p_2 in statement P2, since for such an s no one PE is at P2.

4.2. Reader-optimized version of "Readers and writers".

One advantage of our method is that one can easily check the effect of various changes to a parallel program code. One such modification to Program 4.1, the elimination of the extra checking statement P2, was proposed by Allan Gottlieb. Since the RW-primitive is expected to be heavily used in the Ultracomputer operating system this elimination may gain an essential economy. The modified code is Program 4.2.

Program 4.2

```
COMMENT sem is public; initially sem = N;
        no {...} section accesses sem

     P1: {exterior code; executed for a finite time}
         go to P2 (to read) or to P5 (to write)

     P2: if  F&A(sem,-1) ≥ 1  then go to P4   |
                                              |   read entry
     P3: F&A(sem,1)                           |
         go to P2                             |

     P4: {critical read section; executed for a finite time}

         F&A(sem,1)                           |   read exit
         go to P1


     P5: if  sem ≤ N-1  then go to P5         |
                                              |
     P6: if  F&A(sem,-N) ≥ N  then go to P8   |
                                              |   write entry
     P7: F&A(sem,N)                           |
         go to P5                             |

     P8: {critical write section; executed for a finite time}

         F&A(sem,N)                           |   write exit
         go to P1
```

A summary of the proof of property (RWr) for the obtained program is given below.

This code is a normal program. The program invariant $p_{row} = \{sem = N - n_3 - n_4 - N \times (n_7 + n_8)\}$ supplies the conservative function to eliminate sem from the collapsed program. After such an elimination a reachability tree consisting of 24 metastates, and a RSD consisting of 7 metastates were produced by verifier-2.

Next, the description of the subgraph Δ' of the reachability graph generated by the predicate $\{n_4 = n_8 = 0\}$ was obtained from the RSD. To produce this description the following operations over the text of the RSD were executed: "delete all strings with an occurrence of a given pattern", "replace all occurrences of a given pattern by another pattern". Such patterns were symbols n_i and Pj. (Cf. procedure in section 3.2.) The resulted description consists of 4 (reduced) metastates. The reachability tree, the RSD, and the description of Δ' are given in Appendix.

The following description of all cycles in Δ' was obtained next:

S19 $n_1 + n_2 + n_5 + n_6 + n_7 = N$
$n_7 \geq 1$, $n_1 + n_2 + n_5 \geq 1$, sem = $N \times (1-n_7)$
moves from P2 to P3 lead to S22
moves from P5 to P5 lead to S19

S22 $n_1 + n_2 + n_3 + n_5 + n_6 + n_7 = N$
$n_7 \geq 1$, $n_3 \geq 1$, sem = $N \times (1-n_7) - n_3$
moves from P2 to P3 lead to S22
moves from P3 to P2 lead to S22 (if $n_3 \geq 1$)
or lead to S19 (if $n_3 = 0$)
moves from P5 to P5 lead to S22

It is easy to see that none of the presented cycles satisfies (ILP), since $n_7 \geq 1$ but n_7 PEs at P7 never move.

4.3. Detecting cessation of parallel activity.

Here we analyse the subroutine which detects the situation in which a shared queue is and will remain empty: when all the PEs are trying to delete from an empty queue. The type of applications for which such a detection might be necessary is discussed in [5]. In such an application, multiple PEs, each acting as both a producer and a consumer, use a global queue to buffer data items which they pass among themselves.

Program 4.3

```
COMMENT e is the flag of emptiness of the queue;
        e = 1, if the queue is empty, e = 0, otherwise;
        initially e = 0;
        w is number of PEs waiting to signal
        cessation of activity; initially w = 0.

     P1:  go to P2 (to produce) or go to P3 (to consume)

     P2: (produce an item and insert it into the queue;
          the queue becomes non-empty) e <- 0
          go to P1

     P3:  if e = 1 (the queue is found empty)
          then go to P6 (invoke detection subroutine)

     P4: (the queue is found non-empty;
          if not the last item remains in the queue,
          then delete this item and ) go to P1
          or (the last item remains in the queue) go to P5

     P5: (delete the last item from the queue and
          signal this deletion) e <- 1
          go to P1

     P6:  F&A(w,1)                                  |
                                                    |
     P7:  if e = 1 then go to P9                    |
                                                    |
     P8:  F&A(w,-1)                                 |  detection
          go to P3                                  |  subroutine
                                                    |
     P9:  if  w ≤ N-1  then go to P7                |
                                                    |
     P10: (state T is achieved)                     |
```

We interprete Program 4.3 as follows. The queue empty condition (e = 1) is not sufficient to signify task completion since inserts may still occur even after the empty condition has been raised. Thus, after a queue empty occurs (statement P3) this program increments the counter w (statement P6) to detect a state in which all PEs are trying to delete from an empty queue (we denote this state T). Then w is

compared with the total number N of PEs (statement P9). If w and N are equal, the test in P9 fails and the state T has occurred. If not, the test succeeds and the PE loops until it either finds the queue non-empty (at statement P7), in which case w is decremented (statement P8) and the deletion is retried (starting from statement P3), or until w equals N, in which case state T has occurred.

Note that Program 4.3 approximates (in the sense of section 3.9) the actual queue insertion and deletion subroutines. The complete code of the latter routines is given in [5].

We did not specify statement P10 in this code. The programmer may instruct the parallel computer in any appropriate fashion at P10, although P10 must not contain a "goto" directing to any of the statements P1-P9. One possibility is to call the synchronization routine considered in section 2.12 and then start executing another task.

It must be proved that once a PE achieves statement P10, the cessation of activity is detected. More specifically, the following properties of the above code are to be established:

(CA1) If there is no one item on the queue (i.e. e = 1) for any time $t \geq t_1$, then there exists such a time t_2 that for all $t \geq t_2$ all PEs have detected the cessation of activity (i.e. they have passed P10).

(CA2) If there is a PE which has detected cessation of activity at time t_1 (i.e. this PE has passed P10), then there is no one item on the queue (i.e. e = 1) for all $t \geq t_1$.

This is not a normal program, since a conservative function for counter e does not exist. For example, if N = 2, one PE is at P2 and one PE is at P3, then two states, one with e = 0 and the other with e = 1, are both reachable. Hence e can not be a function of n_i's. Nevertheless the invariant $\{w = n_7 + n_8 + n_9 + n_{10}\}$ provides a conservative function independent of N for the counter w, and the counter e takes only a fixed finite number of values. Therefore this program is almost-normal.

A RSD for this program produced by verifier-2 and the proof of properties (CA1) and (CA2) using this RSD are given in [12].

4.4. "Busy-wait" synchronization.

Program. 2.7 satisfies the following properties:

(SEe) No one PE can enter the next asynchronous section (i.e. P1) while some PE is still in the previous asynchronous section.

(SIe) No more than one PE can execute statement P3 at a time.

(Sr) No one PE can be trapped in the synchronization routine (i.e. at P2) for ever.

(SEe) stands for Synchronization-Exterior-exclusion and (SIe) stands for Synchronization-Interior-exclusion. Both these properties are simular to property (CSe). (Sr) stands for Synchronization-reachability, and is similar to property (CSr).

To prove these properties the skeleton Program 2.8 was used. This code is not a normal program. For example, if N = 2, one PE is at P2 and one PE is at P3, then two states, one with count(1) = 1 and count(2) = 0 and the other with count(1) = 0 and count(2) = 1, are both reachable.

However, we can express count(1) and count(2) as function of n_i's and a two-valued counter q, which is equal modulo two to the number of invocations of this routine. For example

$$\text{count}(1) = \begin{cases} n_2 + n_3, & \text{if } q = 1 \\ 0, & \text{if } q = 0 \end{cases}$$

Initially q = 1, and q changes its value (from 1 to 0 and from 0 to 1) during execution of statements P3 and P6. Note that unlike the example in section 4.3, in which an existing program counter e was taken as the auxiliary counter, in this example the auxiliary counter q did not exist in the original program but was introduced to analyze it.

Acknowledgement

Verifier-2 makes use of a projection algorithm contributed by Robert Thaw [21]. This algorithm will be described in part II of this paper. I would like to thank G. O. Williams and P. J. Teller for reading the manuscript.

Appendix

Notes: To economize on space, branching conditions are omitted, and each group of directing phrases of the form "moves from Pi to Pj lead to Sk" for fixed i, j and k = k1, k2,... are written as a single directing phrase "moves...lead to Sk1 Sk2...".

The value of sem is not specified in these figures for the same reason. The formula sem = $N - n_3 - n_4 - N \times (n_7 + n_8)$ serves in all cases.

There exists a directing phrase, "leading" to S23 in the reachability tree, but the metastate S23 is not presented there, since before S23 was printed verifier-2 had realized that S23 could be eliminated. This does not prevent proving the reachability of all metastates of the RSD.

REACHABILITY TREE

S1 $n_1 = N$
moves from P1 to P2 lead to S2

S2 $n_1 + n_2 = N$
moves from P1 to P5 lead to S3

S3 $n_1 + n_2 + n_5 = N$
moves from P2 to P4 lead to S4
moves from P5 to P6 lead to S5

S4 $n_1 + n_2 + n_4 + n_5 = N$
$n_4 \geq 1$

S5 $n_1 + n_2 + n_5 + n_6 = N$
moves from P2 to P4 lead to S6
moves from P6 to P8 lead to S7

S6 $n_1 + n_2 + n_4 + n_5 + n_6 = N$
$n_4 \geq 1$
moves from P6 to P7 lead to S8

S7 $n_1 + n_2 + n_5 + n_6 + n_8 = N$
$n_8 = 1$
moves from P2 to P3 lead to S9
moves from P6 to P7 lead to S10

S8 $n_1 + n_2 + n_4 + n_5 + n_6 + n_7 = N$
$n_7 \geq 1$
$n_4 \geq 1$
moves from P2 to P3 lead to S11

S9 $n_1 + n_2 + n_3 + n_5 + n_6 + n_8 = N$
$n_8 = 1$
$n_3 \geq 1$
moves from P6 to P7 lead to S12

S10 $n_1 + n_2 + n_5 + n_6 + n_7 + n_8 = N$
$n_8 = 1$
moves from P8 to P1 lead to S13

S11 $n_1 + n_2 + n_3 + n_4 + n_5 + n_6 + n_7 = N$
$n_7 \geq 1$
$n_4 \geq 1$
moves from P4 to P1 lead to S14

S12 $n_1 + n_2 + n_3 + n_5 + n_6 + n_7 + n_8 = N$
$n_8 = 1$
$n_3 \geq 1$
moves from P8 to P1 lead to S15

S13 $n_1 + n_2 + n_5 + n_6 + n_7 + n_8 = N$
$n_8 \leq 1$
$n_7 + n_8 \geq 1$
$n_1 + n_8 \geq 1$
moves from P1 to P2 lead to S16

S14 $n_1 + n_2 + n_3 + n_4 + n_5 + n_6 + n_7 = N$
$n_7 \geq 1$
$n_3 + n_4 \geq 1$
$n_1 + n_4 \geq 1$
moves from P1 to P2 lead to S17

S15 $n_1 + n_2 + n_3 + n_5 + n_6 = N$
$n_3 \geq 1$
$n_1 \geq 1$
moves from P1 to P2 lead to S18

S16 $n_1 + n_2 + n_5 + n_6 + n_7 + n_8 = N$
$n_8 \leq 1$
$n_7 + n_8 \geq 1$
$n_1 + n_2 + n_8 \geq 1$
moves from P1 to P5 lead to S19

S17 $n_1 + n_2 + n_3 + n_4 + n_5 + n_6 + n_7 = N$
$n_7 \geq 1$
$n_3 + n_4 \geq 1$
$n_1 + n_2 + n_4 \geq 1$
moves from P1 to P5 lead to S20

S18 $n_1 + n_2 + n_3 + n_5 + n_6 = N$
$n_3 \geq 1$
$n_1 + n_2 \geq 1$
moves from P1 to P5 lead to S21

S19 $n_1 + n_2 + n_5 + n_6 + n_7 + n_8 = N$
$n_8 \leq 1$
$n_7 + n_8 \geq 1$
$n_1 + n_2 + n_5 + n_8 \geq 1$
moves from P2 to P3 lead to S22

S20 $n_1 + n_2 + n_3 + n_4 + n_5 + n_6 + n_7 = N$
$n_7 \geq 1$
$n_3 + n_4 \geq 1$
$n_1 + n_2 + n_4 + n_5 \geq 1$
moves from P7 to P5 lead to S23

S21 $n_1 + n_2 + n_3 + n_5 + n_6 = N$
$n_3 \geq 1$
$n_1 + n_2 + n_5 \geq 1$
moves from P2 to P4 lead to S24

S22 $n_1 + n_2 + n_3 + n_5 + n_6 + n_7 + n_8 = N$
$n_8 \leq 1$
$n_7 + n_8 \geq 1$
$n_3 \geq 1$

S24 $n_1 + n_2 + n_3 + n_4 + n_5 + n_6 = N$
$n_3 \geq 1$
$n_1 + n_2 + n_4 + n_5 \geq 1$

REACHABILITY SET DESCRIPTION

S5 $n_1 + n_2 + n_5 + n_6 = N$
moves from P1 to P2 lead to S5
moves from P1 to P5 lead to S5
moves from P2 to P4 lead to S6
moves from P5 to P6 lead to S5
moves from P6 to P8 lead to S19

S6 $n_1 + n_2 + n_4 + n_5 + n_6 = N$
$n_4 \geq 1$
moves from P1 to P2 lead to S6
moves from P1 to P5 lead to S6
moves from P2 to P4 lead to S6
moves from P4 to P1 lead to S6
moves from P4 to P1 lead to S5
moves from P5 to P5 lead to S6
moves from P6 to P7 lead to S20

S19 $n_1 + n_2 + n_5 + n_6 + n_7 + n_8 = N$
$n_8 \leq 1$
$n_7 + n_8 \geq 1$
$n_1 + n_2 + n_5 + n_8 \geq 1$
moves from P1 to P2 lead to S19
moves from P1 to P5 lead to S19
moves from P2 to P3 lead to S20 S22
moves from P5 to P5 lead to S19
moves from P6 to P7 lead to S19
moves from P7 to P5 lead to S19
moves from P7 to P5 lead to S5
moves from P8 to P1 lead to S19
moves from P8 to P1 lead to S5

S20 $n_1 + n_2 + n_3 + n_4 + n_5 + n_6 + n_7 = N$
$n_7 \geq 1$
$n_3 + n_4 \geq 1$
$n_1 + n_2 + n_4 + n_5 \geq 1$
moves from P1 to P2 lead to S20 S22
moves from P1 to P5 lead to S20 S22
moves from P2 to P3 lead to S20 S22
moves from P2 to P3 lead to S20 S22
moves from P3 to P2 lead to S20 S22
moves from P3 to P2 lead to S19
moves from P4 to P1 lead to S20 S22
moves from P4 to P1 lead to S19
moves from P5 to P5 lead to S20
moves from P6 to P7 lead to S20 S22
moves from P7 to P5 lead to S20 S22
moves from P7 to P5 lead to S6 S21 S24

S21 $n_1 + n_2 + n_3 + n_5 + n_6 = N$
$n_3 \geq 1$
$n_1 + n_2 + n_5 \geq 1$
moves from P1 to P2 lead to S21 S24
moves from P1 to P5 lead to S21 S24
moves from P2 to P4 lead to S24
moves from P3 to P2 lead to S21 S24
moves from P3 to P2 lead to S5
moves from P5 to P5 lead to S21
moves from P6 to P7 lead to S20 S22

S22 $n_1 + n_2 + n_3 + n_5 + n_6 + n_7 + n_8 = N$
$n_8 \leq 1$
$n_7 + n_8 \geq 1$
$n_3 \geq 1$
moves from P1 to P2 lead to S20 S22
moves from P1 to P5 lead to S20 S22
moves from P2 to P3 lead to S20 S22
moves from P3 to P2 lead to S20 S22
moves from P3 to P2 lead to S19
moves from P5 to P5 lead to S22
moves from P6 to P7 lead to S20 S22
moves from P7 to P5 lead to S20 S22
moves from P7 to P5 lead to S21 S24
moves from P8 to P1 lead to S20 S22
moves from P8 to P1 lead to S21 S24

S24 $n_1 + n_2 + n_3 + n_4 + n_5 + n_6 = N$
$n_3 \geq 1$
$n_1 + n_2 + n_4 + n_5 \geq 1$
moves from P1 to P2 lead to S21 S24
moves from P1 to P5 lead to S21 S24
moves from P2 to P4 lead to S24
moves from P3 to P2 lead to S6 S21 S24
moves from P3 to P2 lead to S5
moves from P4 to P1 lead to S21 S24
moves from P5 to P5 lead to S24
moves from P6 to P7 lead to S20 S22

DESCRIPTION OF THE SUBGRAPH GENERATED BY PREDICATE $p = \{n_4 = n_8 = 0\}$

S5 $n_1 + n_2 + n_5 + n_6 = N$
moves from P1 to P2 lead to S5
moves from P1 to P5 lead to S5
moves from P5 to P6 lead to S5

S19 $n_1 + n_2 + n_5 + n_6 + n_7 = N$
$n_7 \geq 1$
$n_1 + n_2 + n_5 \geq 1$
moves from P1 to P2 lead to S19
moves from P1 to P5 lead to S19
moves from P2 to P3 lead to S22
moves from P5 to P5 lead to S19
moves from P6 to P7 lead to S19
moves from P7 to P5 lead to S19
moves from P7 to P5 lead to S5

S22 $n_1 + n_2 + n_3 + n_5 + n_6 + n_7 = N$
$n_7 \geq 1$
$n_3 \geq 1$
moves from P1 to P2 lead to S22
moves from P1 to P5 lead to S22
moves from P2 to P3 lead to S22
moves from P3 to P2 lead to S22
moves from P3 to P2 lead to S19
moves from P5 to P5 lead to S22
moves from P6 to P7 lead to S22
moves from P7 to P5 lead to S22
moves from P7 to P5 lead to S24

S24 $n_1 + n_2 + n_3 + n_5 + n_6 = N$
$n_3 \geq 1$
$n_1 + n_2 + n_5 \geq 1$
moves from P1 to P2 lead to S24
moves from P1 to P5 lead to S24
moves from P3 to P2 lead to S24
moves from P3 to P2 lead to S5
moves from P5 to P5 lead to S24
moves from P6 to P7 lead to S22

References.

1. Ashcroft, E.A.:. Proving assertions about parallel programs. Journ. Comp. Syst. Sciences 10, pp. 110-135 (1975).

2. Clarke, E.M.: Synthesis of resource invariants for concurrent programs. ACM Trans. on Programming Languages and Systems, Vol.2, No.3, July 1980, pp. 338-358.

3. Courtois, P.J., Heymans, F., and Parnas, D.L.: Concurrent control with 'readers' and 'writers'. Comm. ACM 14, pp. 667-778, 1971.

4. Dijkstra, E.M.: Hierarchial orderings of sequential processes. Acta Informatica, Vol.1, pp. 115-138 (1971).

5. Gottlieb, A., Lubachevsky, B.D., and Rudolph, L.: Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. ACM Trans. on Comp. Lang. Sys., to be published in 1983.

6. Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, K.P., Rudolph, L., and Snir, M.:. The NYU Ultracomputer - designing a MIMD, shared memory parallel machine. Ultracomputer Note #40, Courant Institute, New York University, April, 1982.

7. Hack, M.: The equality problem for vector addition systems is undecidable. Theoretical Comp. Sci., Vol.2 (1976), pp. 77-95.

8. Habermann, A.N.: Synchronization of communicating processes. Comm. ACM, Vol. 15, No. 3 (March 1972), pp. 176-176.

9. Karp, R.M., and Miller, R.E.: Parallel program schemata. J. Comp. Syst. Sci. 3., 147-195 (1969).

10. Kwong, Y.S.: On the absence of livelocks in parallel programs. In: G. Goos, J. Hartmanis (eds.): Semantics of Concurrent Computation. Proc. of the Int. Symp. Lecture Notes in Computer Science 70, Berlin-Heidelberg-New York: Springer 1979, pp. 172-190.

11. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. 3, 125-143 (1977).

12. Lubachevsky, B: Verification of several parallel coordination programs based on descriptions of their reachability sets. Ultracomputer Note #33, Courant Institute, New York University, July, 1981.

13. Owicki, S., and Gries, D.: Verifying properties of parallel programs: an axiomatic approach. Comm. ACM 19, No.5, p. 279-285 (May 1976).

14. Owicki, S., and Gries, D.: An axiomatic proof technique for parallel programs.I. Acta Inf. 6, p. 319-340, (1976).

15. Owicki, S. and Lamport, L.: Proving liveness properties of concurrent programs. ACM Trans. on Prog. Lang. and Syst., 3, vol.4, (July 1982).

16. Pnueli, A.: The temporal semantics of concurrent programs. Theoretical Comp. Sci., Vol.13 (1981), p. 45-60.

17. Rudolph, L.: Software structures for ultraparallel computings. Ph.D. Thesis, Courant Inst., NYU, 1982.

18. Schwartz, J.T.: Ultracomputers. ACM Trans. on Prog. Lang. Sys., 1980, pp. 484-521.

19. Schwartz, R.L., and Melliar-Smith, P.M.: Temporal logic specification of distributed systems. Proc. 2-nd Int. Conf. on Distributed Computing Systems, Paris, April 1981, pp. 446-454.

20. Shaw, A.C.: The logical design of operating systems (1974 ed.). Prentice-Hall.

21. Thau, R.: Private communication.

NYU c.2
Comp. Sci. Dept.
TR-60 Lubachevsky

AUTHOR
An approach to automating

TITLE
the verification of compact

DATE DUE	BORROWER'S NAME

N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012