Automatic Program Construction Techniques

Editors

Alan W. Biermann
Gérard Guiho
Yves Kodratoff

Macmillan Publishing Company
A Division of Macmillan, Inc.
NEW YORK

Collier Macmillan Publishers
LONDON

Macmillan Publishing Company
866 Third Avenue, New York, N.Y. 10022

Collier Macmillan Canada, Inc.

Printed in the United States of America

printing number

1 2 3 4 5 6 7 8 9 10

Library of Congress Cataloging in Publication Data
Main entry under title:

Automatic program construction techniques.

"Outgrowth of a meeting that was held several years ago at the beautiful Centre culturel de Bonas in southern France. The sponsors were the U.S. Army Research and Standardization Group, the French Centre national de la recherche scientifique and the Institut national de recherche en informatique et automatique"--Pref.
Includes bibliographies and index.
1. Automatic programming (Computer science) Addresses, essays, lectures. I. Biermann, Alan W.,

Library of Congress Cataloging in Publication Data

1939- . II. Guiho, Gérard, 1945-
III. Kodratoff, Yves. IV. U.S. Army Research and Standardization Group. V. Centre national de la recherche scientifique (France) VI. Institut national de recherche en informatique et en automatique (France)
QA76.6.A89 1984 001.64'2 83-26817
ISBN 0-02-949070-7

CHAPTER 12

Invariance Proof Methods And Analysis Techniques For Parallel Programs

Patrick Cousot
Université de Metz
Faculté des Sciences
Ile du Saulcy
57045 Metz cedex
France

Radhia Cousot
Centre de Recherche en Informatique de Nancy
France

A. Introduction

We propose a unified approach for the study, comparison and systematic construction of program proof and analysis methods. Our presentation will be mostly informal but the underlying formal theory can be found in Cousot and Cousot [1980b, 1979], and Cousot, P. [1981].

We use discrete state transition systems (Keller [1976], Cousot, P. [1979, 1981]) as abstract models of programs so that our approach is independent of any particular programming language. We use parallel programs with shared variables for illustration purposes.

Our approach is also independent of the particular class of program properties which is considered. For simplicity we only consider invariance properties in this paper. Important properties falling under this category are partial correctness, non termination, absence of run-time errors, deadlock freedom, mutual exclusion, etc.

Since programs are finite descriptions of arbitrarily long and sometimes infinite computations, properties of these computations can only be proved using some inductive reasoning. Hence program proof methods rely upon basic induction principles. For a given class of program properties several different induction principles can be considered. For simplicity, only one basic induction principle will be considered in this paper, which underlies Floyd [1967]'s partial correctness proof method. (A number of different although equivalent induction principles for invariance can be found in Cousot and Cousot [1982].)

All proof methods which rely upon the same induction principle intuitively look similar, but can be difficult to compare in the abstract. We offer a unified view for comparing them. It consists in showing that the verification conditions involved in any of these methods can be obtained by decomposition of the global inductive hypothesis used in the induction principle into an equivalent set of local inductive hypotheses. (Such decompositions can be formalized as connections between lattices (see Cousot and Cousot [1979, 1980b]) and in particular obtained by a cover of the set of states of the program where each local inductive hypothesis holds for a given block of the cover (Cousot [1979]). It is possible to find as many proof methods as such different decompositions. We illustrate only three of them which respectively lead to the Floyd [1967], Owicki and Gries [1976], and Lamport [1977, 1980] invariance proof methods. This approach also provides a framework for systematically constructing new sound and complete proof methods based on unexplored induction principles or decompositions. (See for example Cousot, R. [1981], Cousot and Cousot [1980a]).

Static program flow analysis techniques can be used for discovering semantic properties of programs, that is, for discovering properties of the runtime behavior of programs without actually running them. Such analysis methods consist in solving a fixed point system of equations (by elimination or iteration algorithms) associated with the program to be analyzed (Cousot and Cousot [1977]). In the design of such methods the essential part consists in defining correctly the rules for associating the system of equations with the program. We have shown (Cousot and Cousot [1979]) that they can be derived from the verification conditions of a proof method using an approximate decomposition, hence from a basic induction principle. We illustrate this point of view by generalizing Cousot and Cousot [1976] to parallel programs with shared variables. Another example can be found in Cousot and Cousot [1980b] that generalizes Cousot and Halbwachs [1978] to parallel processes communicating by rendezvous.

B. An Abstract Model of Parallel Programs: Non-Deterministic Transition Systems

An essential step in understanding invariance proof methods consists in considering an abstract model of programs so that irrelevant details can be left unspecified. For that purpose we will consider that a program P defines a dynamic discrete transition system that is a quadruple (S,t,b,e) where:

S	is a set of states,
$t \in (S \times S \rightarrow \{tt,ff\})$	is a transition relation,
$b \in (S \rightarrow \{tt,ff\})$	characterizes entry states,
$e \in (S \rightarrow \{tt,ff\})$	characterizes exit states.

The set S of states is a model of the set of possible data that can be contained in the store(s) on which the program operates. We ignore for the moment the particular structure of the states. In practice a state has several memory components (assigning values to program variables, input and output files, ...) and control components (assigning values to program location counters, ...). Program execution always begins with entry states. The total function b from states into truth values {tt,ff} characterizes entry states. This means that b(s) = tt if and only if state s is an entry state and b(s) = ff otherwise. Program execution properly ends when an exit state is reached. Exit states are characterized by e. The transition relation t specifies the effect of executing an elementary program step. More precisely t(s,s') = tt means that starting in state s and executing one program step can put the program in successor state s'. A sequential program is modeled by a deterministic transition relation since a state s can only have one successor state s', if any. A parallel program is modeled by a non-deterministic transition relation since a state s can have no or several successor states s'. This is because the transition relation is usually defined in terms of arbitrarily choosing an active process and executing one step of that process. Some states s may have no successor (that is t(s,s') = ff for all $s' \in S$), in which case they are called blocking states. For example, a sequential program can be in a blocking state after a run-time error or a parallel program can be in a blocking state because all processes which are not terminated are waiting for some event that never happens.

Example B.1: Defining the semantics of a sequential program by means of a deterministic transition system.

We will consider sequential programs with assignment, conditional and iteration commands. Labels will only be used to designate program points. For simplicity, type and variable declarations are left implicit.

For example, the following program computes 2^n for $n \geqslant 0$:

```
L1:
   P:=1;
L2:
  while N > 0 do
L3:
  N:=N-1;P:=2×P;
L4:
  od;
L5:
```

Let $\Pi = \{li,...,hi\}$ be the set of integers included between the lowest and greatest machine representable integers li and hi. A state $(l,n,p) \in S$ consists of a memory state, that is a pair $(n,p) \in M$ assigning integer values to program variables N,P and of a control state $l \in C$ which is one of the program points, L1,...,L5. Therefore,

$$C = \{L1,...,L5\}$$
$$M = \Pi^2$$
$$S = C \times M$$

Program execution begins at point L1 and ends at point L5 so that

$$b(l,n,p) = [l = L1] \quad \text{characterizes entry states,}$$
$$e(l,n,p) = [l = L5] \quad \text{characterizes exit states.}$$

We define the transition relation t by the following clauses (where $n \in \Pi$ and $p \in \Pi$):

$$(L1,n,p) \xrightarrow{\perp} (L2,n,1) \quad \text{iff} \quad 1 \in \Pi$$
$$(L2,n,p) \xrightarrow{\perp} (L3,n,p) \quad \text{iff} \quad n > 0$$
$$(L2,n,p) \xrightarrow{\perp} (L5,n,p) \quad \text{iff} \quad n \leqslant 0$$
$$(L3,n,p) \xrightarrow{\perp} (L4,n-1,2\times p) \quad \text{iff} \quad (n-1) \in \Pi \text{ and } (2\times p) \in \Pi$$
$$(L4,n,p) \xrightarrow{\perp} (L3,n,p) \quad \text{iff} \quad n > 0$$
$$(L4,n,p) \xrightarrow{\perp} (L5,n,p) \quad \text{iff} \quad n \leqslant 0$$

A clause $[s \xrightarrow{\perp} f(s) \text{ iff } c(s)]$ means that for all $s \in S$, $t(s,f(s)) = tt$ whenever condition $c(s)$ holds.

Starting with $N=2$ and $P=p$, execution of that program leads to the sequence of states $(L1,2,p) \xrightarrow{\perp} (L2,2,1) \xrightarrow{\perp} (L3,2,1) \xrightarrow{\perp} (L4,1,2) \xrightarrow{\perp} (L3,1,2) \xrightarrow{\perp} (L4,0,4) \xrightarrow{\perp} (L5,0,4)$.□

Example B.2: Defining the semantics of a parallel program by means of a non-deterministic transition system.

We consider parallel programs $[\![P_1 |...| P_k]\!]$ which consist of $k>1$ sequential processes $P_1,...,P_k$ executed concurrently. These processes share (implicitly declared) global variables. (If variables need to be local to some process P_i, we will use instead global variables neither used nor modified by the other processes P_j, $j \neq i$.)

It is sometimes necessary that processes have exclusive access to shared global variables. For that purpose we will enclose atomic operations inside square brackets. The execution of such operations is indivisible so that it cannot interfere with the concurrent execution of other processes. For example the program

$$[\![[N:=N+1] \mid [N:=N+1]]\!]$$

will increment N by two, whereas the program

$$[\![[T1:=N];[T1:=T1+1];[N:=T1] \mid [T2:=N];[T2:=T2+1];[N:=T2]]\!]$$

will increment N by one if both processes read the value of N before it is modified by the other process and by two if one process reads the value of N after it has been incremented by the other process.

The following parallel program computes 2^n when $n \geqslant 0$:

```
L0:
  [[
    L11:
      [P1 := 1];
    L12:
      while [N>1] do
    L13:
      [N := N-1; P1 := 2×P1];
    L14:
      od;
    L15:
  |
    L21:
      [P2 := 1];
    L22:
      while [N>1] do
    L23:
       [N := N-1; P2 := 2×P2];
    L24:
      od;
    L25:
  ]];
L1:
  if N = 0 then P := P1×P2 else P := 2×P1×P2 fi;
L2:
```

A state is of the form (c,n,p1,p2,p) where the values n,p1,p2,p of variables N,P1,P2,P belong to $\Pi = \{li,...,hi\}$ and the control state c is either L0, L1, L2 or a pair (l1,l2) of labels, one control location for each of the two processes:

$$C = \{L0,L1,L2\} \cup (\{L11,...,L15\}\times\{L21,...,L25\})$$

$$M = \Pi^4$$
$$S = C \times M$$
$$b(c,n,p1,p2,p) = [c = L0] \quad \text{characterizes entry states}$$
$$e(c,n,p1,p2,p) = [c = L2] \quad \text{characterizes exit states}$$

We define the transition relation t by the following clauses (where $l1 \in \{L11,...,L15\}$; $l2 \in \{L21,...,L25\}$; $n,p1,p2,p \in \Pi$):

(a) $(L0,n,p1,p2,p) \xrightarrow{\perp} ((L11,L21),n,p1,p2,p)$

(b) $((L11,l2),n,p1,p2,p) \xrightarrow{\perp} ((L12,l2),n,1,p2,p)$ iff $1 \in \Pi$

(b) $((L12,l2),n,p1,p2,p) \xrightarrow{\perp} ((L13,l2),n,p1,p2,p)$ iff $n > 1$

(b) $((L12,l2),n,p1,p2,p) \xrightarrow{\perp} ((L15,l2),n,p1,p2,p)$ iff $n \leqslant 1$

(c) (b) $((L13,l2),n,p1,p2,p) \xrightarrow{\perp} ((L14,l2),n\text{-}1,2\times p1,p2,p)$ iff $(n-1) \in \Pi$ and $(2\times p1) \in \Pi$

(b) $((L14,l2),n,p1,p2,p) \xrightarrow{\perp} ((L13,l2),n,p1,p2,p)$ iff $n > 1$

(b) $((L14,l2),n,p1,p2,p) \xrightarrow{\perp} ((L15,l2),n,p1,p2,p)$ iff $n \leqslant 1$

... similar clauses for process 2 ...

(d) $((L15,L25),n,p1,p2,p) \xrightarrow{\perp} (L1,n,p1,p2,p)$

$(L1,n,p1,p2,p) \xrightarrow{\perp} (L2,n,p1,p2,p1\times p2)$ iff $(n=0)$ and $(p1\times p2) \in \Pi$

$(L1,n,p1,p2,p) \xrightarrow{\perp} (L2,n,p1,p2,2\times p1\times p2)$ iff $(n\neq 0)$ and $(2\times p1\times p2) \in \Pi$

On program entry, executions of both processes begin simultaneously (a). Then each process progresses at its own speed independently of the other (b). The concurrent execution of commands in different processes is modelled by an interleaved execution which proceeds as a sequence of discrete steps. In each step a command is selected in only one of the processes and is executed to completion before the same or another process may initiate an elementary command and proceed to complete it. Since execution of atomic operations is indivisible it is modelled by a single transition (c). Notice that since P1 and P2 are not shared we could have split [N:=N-1;Pi:=2×Pi] into [N:=N-1];[Ti:=Pi];[Ti:=2×Ti];[Pi:=Ti]. However the update of N must be indivisible. This can be achieved by any hardware or software mutual exclusion mechanism. The concurrent execution of the two processes ends when both have terminated (d).

A possible execution sequence for N=2 could be:

$(L0,2,p1,p2,p) \xrightarrow{\perp} ((L11,L21),2,p1,p2,p) \xrightarrow{\perp} ((L12,L21),2,1,p2,p) \xrightarrow{\perp}$

$((L13,L21),2,1,p2,p) \xrightarrow{\perp} ((L14,L21),1,2,p2,p) \xrightarrow{\perp} ((L14,L22),1,2,1,p) \xrightarrow{\perp}$

$((L15,L22),1,2,1,p) \xrightarrow{\perp} ((L15,L25),1,2,1,p) \xrightarrow{\perp} (L1,1,2,1,p) \xrightarrow{\perp} (L2,1,2,1,4).$

In the above sequence the value of N at L1 was 1. It can also be 0 if both processes simultaneously test that N>1 when N=2. This is the case in the following execution sequence:

$(L0,2,p1,p2,p) \xrightarrow{1} ((L11,L21),2,p1,p2,p) \xrightarrow{1} ((L12,L21),2,1,p2,p) \xrightarrow{1}$

$((L12,L22),2,1,1,p) \xrightarrow{1} ((L13,L22),2,1,1,p) \xrightarrow{1} ((L13,L23),2,1,1,p) \xrightarrow{1}$

$((L14,L23),1,2,1,p) \xrightarrow{1} ((L15,L23),1,2,1,p) \xrightarrow{1} ((L15,L24),0,2,2,p) \xrightarrow{1}$

$((L15,L25),0,2,2,p) \xrightarrow{1} (L1,0,2,2,p) \xrightarrow{1} (L2,0,2,2,4).$

Notice that the undeterminacy about the values of N and P when both processes end can easily be taken into account to yield the correct result. This solution is certainly less costly than the one which would consist in synchronizing the processes in order to avoid possible simultaneous tests of N. Another solution would consist in having one process iterate $\lfloor n/2 \rfloor$ times and the other $\lceil n/2 \rceil$ times. The drawback of this solution is that its efficiency does depend upon the assumption that both processes are executed at about the same speed. On the contrary, the efficiency of the above parallel program does not depend upon the relative speed of execution of the two processes. Another advantage is that it can be easily generalized to an arbitrary number of processes. □

Abstraction from the above examples is left to the reader. In general, the semantics of a programming language can be defined operationally. This consists in defining the transition system associated with each program of the language by induction on the context-free syntax of programs. (See e.g. Cousot, R. [1981]).

C. Invariance Properties of Parallel Programs

Some properties of programs, such as partial correctness, can be proved without reasoning about the set of sequences of states which represent all possible executions of the program starting from any possible entry state. It is sufficient to reason about the set of states which can be reached during execution. This is because the "time" at which a particular state is reached during execution (if ever), is irrelevant for such invariance proverties.

C.1 Definition: Invariance Property

Let t^* be the reflexive transitive closure of t, that is

$$t^*(s,s') = [\exists\, n \geqslant 1,\ s_1,\ldots,s_n \in S^n \mid s=s_1 \wedge (\forall i \in \{1,\ldots,n-1\},\ t(s_i,s_{i+1})) \wedge s_n=s'].$$

Let $\epsilon \in (S \rightarrow \{tt,ff\})$ and $\sigma \in (S \rightarrow \{tt,ff\})$ be characterizations of initial and final states.

A relation $\Psi \in (S \times S \rightarrow \{tt,ff\})$ is said to be invariant if and only if it is a necessary relation between the initial states and their descendants which are final, that is

$$\forall s,s' \in S,\ [\epsilon(s) \wedge t^*(s,s') \wedge \sigma(s')] \Rightarrow \Psi(s,s').$$

An assertion $\Psi \in (S \rightarrow \{tt,ff\})$ is said to be invariant if and only if it characterizes a super-set of the set of final states that can be reached during some execution started with an initial state, that is

$$\forall s,s' \in S,\ [\epsilon(s) \wedge t^*(s,s') \wedge \sigma(s')] \Rightarrow \Psi(s').$$

C.2 Partial Correctness

Proving that a program is partially correct consists in showing that if execution starts at program entry point with initial values $\underline{x}$ of the variables satisfying some precondition $\phi(\underline{x})$ and terminates with final values $\overline{x}$ of the variables then some relation $\theta(\underline{x},\overline{x})$ should hold between the input values $\underline{x}$ and output values $\overline{x}$ of the variables. This is an invariance property which can be stated as

$$\forall \underline{s},\overline{s} \in S,\ [\epsilon(\underline{s}) \wedge t^*(\underline{s},\overline{s}) \wedge \sigma(\overline{s})] \Rightarrow \Psi(\underline{s},\overline{s}).$$

More precisely, if states $s \in S$ are pairs (c,x) consisting of a control state $c \in C$ and a memory state $x \in M$, b characterizes entry states and e characterizes exit states, partial correctness can be stated as

$$\forall \underline{c},\overline{c} \in C,\ \underline{x},\overline{x} \in M,\ [b(\underline{c},\underline{x}) \wedge t^*((\underline{c},\underline{x}),(\overline{c},\overline{x})) \wedge e(\overline{c},\overline{x})] \Rightarrow [\phi(\underline{x}) \Rightarrow \theta(\underline{x},\overline{x})]$$

Notice that the fact that an exit state $(\overline{c},\overline{x})$ can be reached when execution is started with an entry state $(\underline{c},\underline{x})$ is an hypothesis which is assumed to be true for $\Psi((\underline{c},\underline{x}),(\overline{c},\overline{x})) = [\phi(\underline{x}) \Rightarrow \theta(\underline{x},\overline{x})]$ to hold. Therefore termination is not implied. In particular, any non-terminating program is partially correct since $ff \Rightarrow [\phi(\underline{x}) \Rightarrow \theta(\underline{x},\overline{x})]$.

Example C.2.1: Partial correctness of programs B.1 and B.2.

We will prove that programs B.1 and B.2 are partially correct with respect to $\phi(\underline{n},\underline{p}) = [\underline{n} \geqslant 0]$ and $\theta((\underline{n},\underline{p}),(\overline{n},\overline{p})) = [\overline{p} = 2^{\underline{n}}]$. □

C.3 Non termination

A program never terminates if and only if any state which can be reached during execution is not an exit state and has at least one successor state. Non-termination is also an invariance property where ϵ characterizes entry states, σ is identically true and $\psi(\overline{s})$ holds if and only if $\overline{s}$ is neither an exit nor a blocking state.

$$\forall \underline{s},\overline{s} \in S,\ [b(\underline{s}) \wedge t^*(\underline{s},\overline{s})] \Rightarrow [\neg e(\overline{s}) \wedge (\exists s \in S \mid t(\overline{s},s))]$$

C.4 Absence of Run-time Errors

Absence of run-time errors is also an invariance property. It means that whenever a state which is not an exit state is reached during execution, a next computation step is possible without causing a run-time error (such as division by zero, arithmetic overflow, subscript out of range, ...).

Example C.4.1: Clean behavior of program B.1.

For each label l of program B.1, let us formulate a necessary and sufficient condition $\gamma_l(n,p)$ which guarantees that execution of the program commands labelled l in memory state (n,p) will not cause a run-time error:

$$\gamma_1(n,p) = [li \leqslant 1 \leqslant hi]$$

$$\gamma_2(n,p) = \gamma_4(n,p) = \gamma_5(n,p) = tt$$

$$\gamma_3(n,p) = [li \leqslant n-1 \leqslant hi \wedge li \leqslant 2 \times p \leqslant hi]$$

For program B.1 the condition that all integers between 0 and $2^{\underline{n}}$ are machine representable is sufficient to avoid run-time errors. This can be stated as

$$[b(\underline{l},\underline{n},\underline{p}) \wedge t^*((\underline{l},\underline{n},\underline{p}),(l,n,p))] \Rightarrow \Psi((\underline{l},\underline{n},\underline{p}),(l,n,p))$$

where

$$\Psi((\underline{l},\underline{n},\underline{p}),(l,n,p)) = [\,[0 \leqslant \underline{n} \wedge li \leqslant 0 \leqslant 2^{\underline{n}} \leqslant hi] \Rightarrow \gamma_l(n,p)\,] \,.\,\square$$

C.5 Global and Local Invariants

Let P be a program with states $S = C \times M$. A global invariant $\gamma \in (M \rightarrow \{tt,ff\})$ is a predicate on memory states which is always true during execution:

$$\forall \underline{s} \in S,\ c \in C,\ x \in M,\ [b(\underline{s}) \wedge t^*(\underline{s},(c,x))] \Rightarrow \gamma(x).$$

A predicate $\delta \in (M \rightarrow \{tt,ff\})$ on memory states which holds whenever control is at program points $l \in L$, where $L \subseteq C$, is called an invariant local to L:

$$\forall \underline{s} \in S,\ c \in C,\ x \in M,\ [b(\underline{s}) \wedge t^*(\underline{s},(c,x))] \Rightarrow [(c \in L) \Rightarrow \delta(x)].$$

Example C.5.1: Using program flow analysis algorithms for generating local invariants.

Some program analysis techniques, such as Cousot and Halbwachs [1978], can be used for automatic computation of local invariants of programs. Since the strongest set of local invariants is not computable, only approximate results can be automatically obtained. The invariant $\delta(x)$ associated with program points $l \in L$ is approximate in the sense that it is correct:

$$[b(\underline{s}) \wedge t^*(\underline{s},(l,x))] \Rightarrow \delta(x)$$

but does not provide full information, since we may have

$$[b(\underline{s}) \wedge t^*(\underline{s},(l,x))] \not\equiv \delta(x) \quad \square$$

C.6 Absence of Global Deadlocks

Parallel processes may need to be synchronized for the concurrent access of shared resources. For example a process P_i may be willing to use a common resource, which can only be used by one process at the same time, and which is currently being used by some other process P_j, $j \neq i$. Then P_i has to be blocked temporarily and to wait until this resource is released by process P_j. If several processes are waiting it may be necessary to specify the order in which waiting processes will be allowed to use the common resource.

Because of programming errors, it may happen that all processes are blocked permanently so that there is no way to recover. The absence of such global deadlocks is an invariance property. (It may also happen that some subset of the processes in a program is blocked while the other processes remain permanently active. The absence of such individual starvations or livelock is not an invariance property.)

For illustration purposes, we will use conditional critical regions as the synchronization tool. When a process attempts to execute a command

await [B then C]

it is delayed until the condition B is true. Then the command C is executed as an atomic action, the evaluation of B to true and execution of C being indivisible. Command C cannot contain a nested await command. If two or more processes are waiting for the same condition B, any one may be allowed to proceed when B becomes true while the others continue waiting. When invariance properties are considered the order in which waiting processes are scheduled is often irrelevant.

Let us consider a parallel program $[\![P_1 \,|..|\, P_k]\!]$. The corresponding states are of the form $((l_1,...,l_k),x)$ where each l_i is a location of process P_i and x the memory state of the shared variables X. Let W_i be the set of waiting locations of process P_i so that P_i contains await commands

$$Lij\text{:await } [B(Lij)(X) \text{ then } C] \text{ , for } Lij \in W_i.$$

Let Lie be the exit location of process P_i. We define W as the set of control states $(l_1,...,l_k)$ corresponding to waiting or exit locations, not all of them being exit locations. Formally

$$W = \{(l_1,...,l_k) \mid \forall i \in \{1,...,k\}, l_i \in W_i \cup \{Lie\}\} - \{(L1e,...,Lke)\}.$$

A *blocking state* is a state where not all processes have terminated and all of the processes that have not yet terminated are delayed at an await (because the corresponding condition evaluates to false).

Formally a blocking state is characterized by $\beta \in (S \rightarrow \{tt,ff\})$ such that

$$\beta((l_1,...,l_k),x) = [(l_1,...,l_k) \in W \wedge (\forall i \in [1,k], (l_i \neq Lie) \Rightarrow \neg B(l_i)(x))] .$$

A *sufficient condition ensuring absence of global deadlocks* is that all states that can be reached during execution are not blocking states. This invariance property can be stated as

$$\forall \underline{s},s \in S, \ [b(\underline{s}) \wedge t^*(\underline{s},s)] \Rightarrow \neg\beta(s)$$

C.7 Mutual Exclusion

Let P be a parallel program $[\![P_1 |...| P_k]\!]$ with states of the form $((l_1,...,l_k),x)$ where each l_i is a label of process P_i and x is the memory state of the shared variables. Two statements labelled L_i and L_j in processes P_i and P_j, i, j $\in \{1,...,k\}$ are mutually exclusive if they cannot be executed at the same time. This invariance property can be formulated as:

$$\forall \underline{s},((l_1,...,l_k),x) \in S, \ [b(\underline{s}) \wedge t^*(\underline{s},((l_1,...,l_k),x))] \Rightarrow [\neg ((l_i=L_i) \wedge (l_j=L_j))]$$

D. The Basic Sound and Complete Induction Principle for Proving Invariance Properties of Programs

We now begin to introduce our mathematical approach for constructing invariance proof methods. This study is abstract in that by considering a general model of programs (dynamic discrete transition systems) we are not bound to particular programming language features. Also by considering an abstract class of program properties (invariance properties) the study is independent of which particular property in the class is considered. In this paragraph we state the general induction principle underlying almost all methods for proving invariance properties. Next we will explain how particular methods can be derived from this induction principle.

For proving that a program property Ψ is invariant, one usually has to guess a stronger property I which is shown to hold for all descendants of the initial states (a), (b) and to imply Ψ for final states (c). The proof that the inductive invariant I holds for any possible descendant of any initial state is by induction (on the minimal number n of computation steps until execution reaches this descendant of the initial state). The basis (a) consists in proving that the inductive invariant I holds for initial states (that is after n=0 computation step). The induction step (b) consists in proving that if the inductive invariant I holds for some state s' (which is reachable from the initial state by $n \geqslant 0$ computation steps) then I also holds for all possible successors s of that state s' (successors s, which are therefore reachable from the initial states by n+1 computation steps). By induction on n, all descendants of initial states satisfy I. In particular (c), since I implies Ψ for final states, Ψ holds if and when such a final state is reached during execution.

In the following theorem we give a very general formulation of the above invariance proof method using the transition systems framework. In the proof of this theorem we formally rephrase the above soundness (correctness, consistency) argument. We also add a semantic completeness argument (showing that if a property Ψ is invariant, then this can be proved using the general induction principle).

THEOREM D.1

$$[\exists\, I \in (S\times S \rightarrow \{tt,ff\}) \mid \forall \underline{s},s,\bar{s} \in S,$$

$$\begin{array}{ll} (a) & \epsilon(\underline{s}) \Rightarrow I(\underline{s},\underline{s}) \\ \wedge & \\ (b) & [\exists s' \in S \mid I(\underline{s},s') \wedge t(s',s)] \Rightarrow I(\underline{s},s) \\ \wedge & \\ (c) & [I(\underline{s},\bar{s}) \wedge \sigma(\bar{s})] \Rightarrow \Psi(\underline{s},\bar{s}) \qquad] \end{array}$$

$$\Leftrightarrow$$

$$[\forall \underline{s},\bar{s} \in S,\ [\epsilon(\underline{s}) \wedge t^*(\underline{s},\bar{s}) \wedge \sigma(\bar{s})] \Rightarrow \Psi(\underline{s},\bar{s})]$$

Proof: For the *soundness* proof ($\Rightarrow$) we show by recurrence on n that

$$[\forall\, n \geqslant 1,\ s_1,\ldots,s_n \in S,\ [\epsilon(s_1) \wedge (\forall i \in \{1,\ldots,n-1\},\ t(s_i,s_{i+1})] \Rightarrow I(s_1,s_n)].$$

We use (a) for the basis $n=1$ and (b) for the induction step $n>1$. From this lemma we conclude that $[\epsilon(\underline{s}) \wedge t^*(\underline{s},\bar{s}) \wedge \sigma(\bar{s})] \Rightarrow [I(\underline{s},\bar{s}) \wedge \sigma(\bar{s})]$ which according to (c) implies $\Psi(\underline{s},\bar{s})$.

The *completeness* proof ($\Leftarrow$) is also very simple since we can choose $I(\underline{s},s) = [\epsilon(\underline{s}) \wedge t^*(\underline{s},s)]$ so that (a) and (b) follow from the definition of the reflexive transitive closure whereas (c) follows from the hypothesis $[\epsilon(\underline{s}) \wedge t^*(\underline{s},\bar{s}) \wedge \sigma(\bar{s})] \Rightarrow \Psi(\underline{s},\bar{s})$. □

The invariance property is sometimes not a relation between initial and final states but an assertion on final states. In this case we can use the following induction principle, the soundness and correctness proofs of which are easily derived from the above theorem. (A version of this induction principle was originally proposed by Keller [1976]):

Corollary D.2

$$[\exists\, i \in (S \rightarrow \{tt,ff\}) \mid \forall\, \underline{s},s,\bar{s} \in S,$$

$$\begin{array}{ll} (a) & \epsilon(\underline{s}) \Rightarrow i(\underline{s}) \\ \wedge & \\ (b) & [\exists\, s' \in S \mid i(s') \wedge t(s',s)] \Rightarrow i(s) \\ \wedge & \\ (c) & [i(\bar{s}) \wedge \sigma(\bar{s})] \Rightarrow \psi(\bar{s}) \qquad] \end{array}$$

$$\Leftrightarrow$$

$$[\forall\, \underline{s},\bar{s} \in S,\ [\epsilon(\underline{s}) \wedge t^*(\underline{s},\bar{s}) \wedge \sigma(\bar{s})] \Rightarrow \psi(\bar{s})]$$

Proof: The *soundness* proof ($\Rightarrow$) consists in defining $I(\underline{s},s) = i(s)$, $\Psi(\underline{s},\bar{s}) = \psi(\bar{s})$ and applying theorem D.1. The *completeness* proof ($\Leftarrow$) consists in proving that if $I(\underline{s},s)$ satisfies conditions D.1 (a)−(c) then $i(s) = [\exists\, \underline{s} \in S \mid \epsilon(\underline{s}) \wedge I(\underline{s},s)]$ satisfies conditions D.2 (a)−(c). □

Example D.3: Proving the partial correctness of a parallel program by direct application of the basic induction principle.

The program

$$[[L11: [N:=N+1];L12: \mid L21: [N:=N+1];L22:]]$$

defines a non-deterministic transition system (S,t,b,e) such that

$S = \{L11,L12\}\times\{L21,L22\}\times\Pi$	states
$b(l1,l2,n) = [l1=L11 \wedge l2=L21]$	entry states
$e(l1,l2,n) = [l1=L12 \wedge l2=L22]$	exit states
$(L11,l2,n) \xrightarrow{t} (L12,l2,n+1)$ iff $(n+1) \in \Pi$	transition relation
$(l1,L21,n) \xrightarrow{t} (l1,L22,n+1)$ iff $(n+1) \in \Pi$	

Let us prove that if execution of that program begins with N=0 and happens to end then N=2. This partial correctness property can be formulated as

$$\forall \underline{s},\bar{s} \in S,\ [\epsilon(\underline{s}) \wedge t^*(\underline{s},\bar{s}) \wedge \sigma(\bar{s})] \Rightarrow \psi(\bar{s})$$

where

$\epsilon(l1,l2,n) = [b(l1,l2,n) \wedge n=0]$	input specification
$\sigma(l1,l2,n) = e(l1,l2,n)$	
$\psi(l1,l2,n) = [n=2]$.	output specification

This can be proved using the following inductive assertion:

$$\begin{aligned} i(l1,l2,n) = [&(l1=L11 \wedge l2=L21 \wedge n=0) \vee (l1=L11 \wedge l2=L22 \wedge n=1) \\ &\vee (l1=L12 \wedge l2=L21 \wedge n=1) \vee (l1=L12 \wedge l2=L22 \wedge n=2)] \end{aligned}$$

which, as can easily be checked by the reader, satisfies conditions D.2 (a) – (c). □

Readers familiar with fixpoint theory can consult Cousot, P. [1981] where it is shown that the *invariants can be defined as fixpoints of predicate transformers.* In Cousot and Cousot [1981], *other equivalent induction principles are derived from the above ones,* and this leads to the construction of new invariance proof methods.

E. Design of a Proof Method by Decomposition of the Global Invariant of an Induction Principle into a Set of Local Invariants

We now informally explain how practical invariance proof methods can be constructively derived from induction principles D.1 and D.2. The essential idea is to provide for a standard decomposition of the global inductive invariant I into a (logically equivalent) set of local invariants $\{Q_l \mid l \in L\}$, each one holding when control is at some points of the program. Then the verification conditions D.1 (a) – (c) or D.2 (a) – (c) can be

decomposed into a conjunction of simpler verification conditions, each one corresponding to a basic command of the program and each one involving only some of the local invariants Q_l.

Example E.1: The standard decomposition for sequential programs.

Naur [1966], Floyd [1967] and Hoare [1969]'s partial correctness proof method is applicable to sequential programs. A local inductive invariant on memory states Q_l is associated with each program point l. The verification conditions ensure that when execution reaches some program point k which is immediately followed by program point l, then the assumption that Q_k is true when control is at point k implies that Q_l must be true if and when control reaches program point l.

The verification conditions for proving the partial correctness of program B.1 are (for all $\underline{n},n,\underline{p},p,p' \in \Pi$):

(a) $[\underline{n} \geqslant 0] \Rightarrow Q_1(\underline{n},\underline{n},\underline{p})$

(b) $[Q_1(\underline{n},n,p') \wedge 1 \in \Pi \wedge p=1] \Rightarrow Q_2(\underline{n},n,p)$

$[Q_2(\underline{n},n,p) \wedge n>0] \Rightarrow Q_3(\underline{n},n,p)$

$[Q_2(\underline{n},n,p) \wedge n \leqslant 0] \Rightarrow Q_5(\underline{n},n,p)$

$[Q_3(\underline{n},n',p') \wedge (n'-1) \in \Pi \wedge n=(n'-1) \wedge (2\times p') \in \Pi \wedge p=2\times p'] \Rightarrow Q_4(\underline{n},n,p)$

$[Q_4(\underline{n},n,p) \wedge n>0] \Rightarrow Q_3(\underline{n},n,p)$

$[Q_4(\underline{n},n,p) \wedge n \leqslant 0] \Rightarrow Q_5(\underline{n},n,p)$

(c) $Q_5(\underline{n},n,p) \Rightarrow [p = 2^{\underline{n}}]$

Observe that by substitutions we could have eliminated Q_2 and Q_4, keeping only the loop invariant Q_3. This leads to Floyd's method.

The reader can check that the following local invariants satisfy the above verification conditions:

$Q_1(\underline{n},n,p) = [\underline{n} = n \geqslant 0]$

$Q_2(\underline{n},n,p) = [\underline{n} = n \geqslant 0 \wedge p=1]$

$Q_3(\underline{n},n,p) = [n>0 \wedge p=2^{\underline{n}-n}]$

$Q_4(\underline{n},n,p) = [n \geqslant 0 \wedge p=2^{\underline{n}-n}]$

$Q_5(\underline{n},n,p) = [n=0 \wedge p=2^{\underline{n}}]$

In order to understand how this partial correctness proof method can be constructively derived from induction principle D.1, let us define

$$\epsilon(\underline{l},\underline{n},\underline{p}) = [\underline{l}=L1 \wedge \underline{n} \geqslant 0]$$

$$I((\underline{l},\underline{n},\underline{p}),(l,n,p)) = \bigvee_{k=1}^{5} [l=Lk \wedge Q_k(\underline{n},n,p)]$$

Then verification D.1(a) which was

$$(\epsilon(\underline{l},\underline{n},\underline{p}) \Rightarrow I((\underline{l},\underline{n},\underline{p}),(\underline{l},\underline{n},\underline{p})))$$

can be simplified into condition E.1.(a) as follows:

$$\equiv ([\underline{l}=L1 \wedge \underline{n} \geqslant 0] \Rightarrow \bigvee_{k=1}^{5} [\underline{l}=Lk \wedge Q_k(\underline{n},\underline{n},\underline{p})])$$

$$\equiv ([\underline{n} \geqslant 0] \Rightarrow \bigvee_{k=1}^{5} [L1=Lk \wedge Q_k(\underline{n},\underline{n},\underline{p})])$$

$$\equiv ([\underline{n} \geqslant 0] \Rightarrow Q_1(\underline{n},\underline{n},\underline{p}))$$

The same way, condition D.1.(b):

$$[I(\underline{s},s') \wedge t(s',s)] \Rightarrow I(\underline{s},s)$$

can be written as a conjunction of five conditions for $k=1,\ldots,5$:

$$[I((\underline{l},\underline{n},\underline{p}),(Lk,n',p')) \wedge t((Lk,n',p'),(l,n,p))] \Rightarrow I((\underline{l},\underline{n},\underline{p}),(l,n,p))$$

Replacing I and t by their definitions further simplifications lead to the verification conditions E.1.(b). Finally E.1.(c) is equivalent to D.1.(c) where:

$$\sigma(\bar{l},\bar{n},\bar{p}) = [\bar{l} = L5]$$

$$\Psi((\underline{l},\underline{n},\underline{p}),(\bar{l},\bar{n},\bar{p})) = [\bar{p} = 2^{\underline{n}}] \ . \ \square$$

More generally, observe that the basic induction principles D.1 and D.2 have verification conditions of the form:

$$(\alpha) \quad (\exists I \in A \mid V(I))$$

Invariance proof methods apply induction principles D1 or D2 indirectly in that one uses other verification conditions of the form

$$(\beta) \quad (\exists Q' \in A' \mid V'(Q')).$$

Using (β) instead of (α) is sound iff $(\beta) \Rightarrow (\alpha)$, complete iff $(\alpha) \Rightarrow (\beta)$ and equivalent iff $(\alpha) <=> (\beta)$.

In practice we can establish a correspondence between I and Q by means of a pair of functions $\rho \in [A \rightarrow A']$ and $\rho' \in [A' \rightarrow A]$ so that a sufficient soundness condition is

$$\forall Q' \in A', \ V'(Q') \Rightarrow V(\rho'(Q'))$$

and a sufficient completeness condition is

$$\forall I \in A, \ V(I) \Rightarrow V'(\rho(I)).$$

Example E.2: The standard decomposition for sequential programs leads to sound and complete proof methods.

Coming back to the partial correctness proof method which we illustrated by example E.1 we had:

$$C = \{L1,...,L5\}$$

$$S = C \times \Pi^2$$

$$A = (S^2 \rightarrow \{tt,ff\})$$

$$A' = \prod_{l \in C} (\Pi^3 \rightarrow \{tt,ff\})$$

That is $Q \in A'$ was a vector of assertions Q_l, $l \in C$ on $(\underline{n},n,p) \in \Pi^3$. The correspondence between A and A' was defined by $\rho' \in [A' \rightarrow A]$ such that:

$$\rho(Q)((\underline{l},\underline{n},\underline{p}),(l,n,p)) = \bigvee_{k=1}^{5} [l=Lk \wedge Q_k(\underline{n},n,p)]$$

and $\rho \in [A \rightarrow A']$ such that:

$$\rho(I) = (Q_1,...,Q_5)$$

where

$$Q_l(\underline{n},n,p) = [\exists \underline{l} \in C, \underline{n},\underline{p} \in \Pi \mid I((\underline{l},\underline{n},\underline{p}),(l,n,p))].$$

This correspondence formally defines what is usually explained as "$Q_l(\underline{n},n,p)$ relates the initial value $\underline{n}$ of N and the current values n,p of variables N,P when control is at program point l".

Notice that ρ is one-to-one-onto and ρ' is its inverse. Since

$$\forall Q \in A',\ V'(Q) = V(\rho'(Q))$$

the method is sound. Moreover,

$$\forall I \in A,\ V(I) \Rightarrow V(\rho'(\rho(I))) = V'(\rho(I)),$$

so that the method is also complete. □

In Cousot and Cousot [1980] we have shown that A and A' can be chosen as *complete lattices* and the pair (ρ,ρ') as a *Galois connection* between these lattices.

Cousot, P. [1981] proposes a systematic method for constructing the set of local invariants A' and the corresponding pair (ρ,ρ') using a *cover of the set* S *of states* of the transition system (S,t,b,e) defined by the operational semantics of a program. There, each local invariant is defined so as to be isomorphic with the restriction of the global inductive invariant to the states belonging to some block of the cover. For example, the decomposition leading to Floyd's method for proving partial correctness of sequential programs (see examples 6.1, 6.2) has been derived in this way, using a partition of the set of states such that states belonging to a given block of the partition all correspond to the same control point of the program.

However our idea of using connections between the lattices A and A' which induce a connection between the predicate transformers corresponding to the verification conditions V and V' (which goes back to Cousot

and Cousot [1976]) is more general in that it is suitable for reasoning about program proof methods (where (A,V) and (A',V') have to be equivalent) and also for reasoning about mechanizable hence fundamentally incomplete program analysis methods (Cousot and Cousot [1979]).

F. Two Invariance Proof Methods for Parallel Programs

We now present two methods for proving invariance properties of parallel programs. Both are derived from induction principle D.2 but using different decompositions of the global invariant involved in this induction principle. The first decomposition consists in associating a local invariant about memory and control states with each point of each process of the program. Choosing this decomposition we obtain Owicki and Gries [1976] (up to the use of auxiliary variables for simulating control states) and Lamport [1977] invariance proof methods. The second decomposition of the global program invariant consists in associating a global process invariant on control and memory states with each process of the program. Choosing this decomposition leads to the Lamport [1980] invariance proof method. We have chosen these two decompositions on purpose, in order to study from a unified point of view two classical methods which are intuitively understood as variations on Floyd's basic method of invariants but are difficult to compare because dissimilar formalisms are used for assertion languages and for the presentation of verification conditions.

F.1 Decomposition of the Global Program Invariant Leading to Owicki and Gries [1976]-Lamport [1977] Proof Method

F.1.1 Decomposition

Let us consider a parallel program $[\![P_1 |...| P_k]\!]$ with memory states M and control states Ci for each process P_i, i = 1, ...,k. A global invariant $I \in (C1 \times \cdots \times Ck \times M \rightarrow \{tt,ff\})$ can be expressed as a conjunction of local invariants $Q_{il} \in (C1 \times \cdots \times Ci-1 \times Ci+1 \times \cdots \times Ck \times M \rightarrow \{tt,ff\})$ on control states (of processes $P_j, j \neq i$) and memory states. A local invariant Q_{il} is attached to each program point $l \in Ci$ of each process P_i, i=1,...,k. More precisely,

$$I(c1,...,ck,x) = \bigwedge_{i=1}^{k} \bigwedge_{l \in C_i} [(c_i = l) \Rightarrow Q_{il}(c_1,...,c_{i-1},c_{i+1},...,c_k, x)]$$

and

$$Q_{il}(c_1,...,c_{i-1},c_{i+1},c_k,x) = I(c_1,...,c_{i-1},l,c_{i+1},...,c_k,x).$$

Example F.1.1.1: The global invariant

$$\begin{aligned} i(c1,c2,n) = [&(c1=1 \wedge c2=3 \wedge n=0) \vee (c1=1 \wedge c2=4 \wedge n=2) \vee \\ &(c1=2 \wedge c2=3 \wedge n=1) \vee (c1=2 \wedge c2=4 \wedge n=3)] \end{aligned}$$

of the parallel program

$$[\![\ 1:[N:=N+1];\ 2: \mid 3:\ [N:=N+2];\ 4:\ ]\!]$$

can be expressed by the set of local invariants

$$Q_1(c2,n) = [(c2=3 \wedge n=0) \vee (c2=4 \wedge n=2)]$$
$$Q_2(c2,n) = [(c2=3 \wedge n=1) \vee (c2=4 \wedge n=3)]$$
$$Q_3(c1,n) = [(c1=1 \wedge n=0) \vee (c1=2 \wedge n=1)]$$
$$Q_4(c1,n) = [(c1=1 \wedge n=2) \vee (c1=2 \wedge n=3)]$$

Notice that if program point l belongs to process i then Q_l holds when control is at l in process i (wherever control can be in the other processes). Therefore these local invariants can be interspread at appropriate places in the program text, with the interpretation that Q_l is a valid comment at program point l . □

F.1.2 Derivation of the verification conditions

Verification condition D.1(b), which for simplicity we denote $I \wedge t => I$ can be decomposed into a conjunction of verification conditions

$$I(c'_1,...,c'_k,x') \wedge t((c'_1,...,c'_k,x'),(c_1,...,c_{i-1},l,c_{i+1},...,c_k,x)) => Q_{il}(c_1,...,c_{i-1},c_{i+1},...,c_k,x)$$

for each process $i \in \{1, ...,k\}$ and program point $l \in C_i$ of that process. Since t is defined as a disjunction of cases, one for each elementary command of the program, the above verification condition can be further decomposed. When the transition corresponds to execution of a command of process i we get verification conditions corresponding to the sequential proof (of process i regarded as an independent sequential program):

$$I(c_1,...,c_{i-1},l',c_{i+1},...,c_k,x') \wedge t((c_1,...,c_{i-1},l',c_{i+1},...,c_k,x'),(c_1,...,c_{i-1},l,c_{i+1},...,c_k,x))$$
$$=> Q_{il}(c_1,...,c_{i-1},c_{i+1},...,c_k,x)$$

When the transition corresponds to execution of a command of process $j \neq i$, we get verification conditions of the form:

$$I(c_1,...,c_{j-1},l',c_{j+1},...,c_{i-1},l,c_{i+1},...,c_k,x') \wedge t((c_1,...,l',...,l,...,c_k,x'),(c_1,...,l'',...,l,...,c_k,x))$$
$$=> Q_{il}(c_1,...,c_{j-1},l'',c_{j+1},...,c_{i-1},c_{i+1},...,c_k,x)$$

which consists in proving that the local invariants of each process are left invariantly true under parallel execution of the other processes. These verification conditions were termed "interference freeness checks" by Owicki and Gries [1976] and "monotonicity conditions" by Lamport [1977]. However, these authors did not exactly propose the above verification conditions but instead the following simpler (and stronger) ones:

– sequential proof:

$$Q_{il'}(c_1,...,c_{i-1},c_{i+1},...,c_k,x') \wedge t((c_1,...,c_{i-1},l',c_{i+1},...,c_k,x'),(c_1,...,c_{i-1},l,c_{i+1},...,c_k,x))$$
$$=> Q_{il}(c_1,...,c_{i-1},c_{i+1},...,c_k,x)$$

– interference freeness checks:

$$Q_{il}(c_1,\ldots,c_{j-1},l',c_{j+1},\ldots,c_{i-1},c_{i+1},\ldots,c_k,x') \wedge Q_{jl'}(c_1,\ldots,c_{j-1},c_{j+1},\ldots,c_{i-1},l,c_{i+1},\ldots,c_k,x')$$

$$\wedge\ t((c_1,\ldots,l',\ldots,l,\ldots,c_k,x'),(c_1,\ldots,l'',\ldots,l,\ldots,c_k,x)) \Rightarrow Q_{il}(c_1,\ldots,c_{j-1},l'',c_{j+1},\ldots,c_{i-1},c_{i+1},\ldots,c_k,x)$$

These verification conditions are obviously sufficient since they imply the above ones

(because $I = \bigwedge_i \bigwedge_l Q_{il}$).

Example F.1.2.1:

The verification conditions corresponding to program F.1.1.1 that is:

$$[\![\ 1\colon [N:=N+1];\ 2\colon |\ 3\colon [N:=N+2];\ 4\colon]\!]$$

are

(a) initialization:

$\phi(\underline{n}) \Rightarrow [Q_1(3,\underline{n}) \wedge Q_3(1,\underline{n})]$, where ϕ is the input specification

(b) induction step:

– sequential proof of process 1:

$[Q_1(c2,n') \wedge (n'+1) \in \Pi \wedge n=n'+1] \Rightarrow Q_2(c2,n)$

– absence of interference of the proof of process 1 with process 2:

$[Q_1(3,n') \wedge Q_3(1,n') \wedge (n'+2) \in \Pi \wedge n=n'+2] \Rightarrow Q_1(4,n)$

$[Q_2(3,n') \wedge Q_3(1,n') \wedge (n'+2) \in \Pi \wedge n=n'+2] \Rightarrow Q_2(4,n)$

– sequential proof of process 2:

$[Q_3(c1,n') \wedge (n'+2) \in \Pi \wedge n=n'+2] \Rightarrow Q_4(c1,n)$

– absence of interference of the proof of process 2 with process 1:

$[Q_3(1,n') \wedge Q_1(3,n') \wedge (n'+1) \in \Pi \wedge n=n'+1] \Rightarrow Q_3(2,n)$

$[Q_4(1,n') \wedge Q_1(3,n') \wedge (n'+1) \in \Pi \wedge n=n'+1] \Rightarrow Q_4(2,n)$

(c) finalization:

$[Q_2(4,\bar{n}) \wedge Q_4(2,\bar{n})] \Rightarrow \theta(\bar{n})$, where θ is the output specification.□

If we define

$$\rho'(Q) = I \text{ iff } I(c_1,...,c_k,x) = \bigwedge_{i=1}^{k} \bigwedge_{l \in C_i} [(c_i=l) => Q_{il}(c_1,...,c_{i-1},c_{i+1},...,c_k,x)]$$

and

$$\rho(I) = Q \text{ iff } Q_{il}(c_1,...,c_{i-1},c_{i+1},...,c_k,x) = I(c_1,...,c_{i-1},l,c_{i+1},...,c_k,x)$$

we have established a formal correspondence between induction principle D.2 and the Owicki-Lamport invariance proof method. We have informally proved that the verification conditions proposed by Owicki-Lamport are sound (i.e. using the notations of paragraph E, that $V'(Q) => V(\rho'(Q))$. The sufficient completeness condition ($V(I) => V'(\rho(I))$) can also be checked by the reader. Intuitively, this condition is satisfied because each local invariant Q_{il} can always be made strong enough so as to exactly describe the possible states of the whole program when process i is at point l.

F.1.3 Example

Let us prove that program B.2 is partially correct (according to definition C.2.1). Instead of using induction principle D.2 which underlies the Owicki and Gries' method we use D.1 so as to be able to relate the current value n of variable N to its initial value $\underline{n}$. (Owicki and Gries would instead introduce an auxiliary variable in order to memorize the initial value of N).

Since both processes are symmetric we need only reason about process 1. We will prove that the relation

$$Inv(\underline{n},c2,n,p1,p2) = [(c2=L21 \wedge p1=2^{\underline{n}-n}) \vee (c2 \neq L21 \wedge p1 \times p2=2^{\underline{n}-n})]$$

is invariant in process 1 after initialization of variable P1. To prove this we will show that the invariant remains true after execution of any command of process 1 and that it is not invalidated by execution of some command of process 2. Since partial correctness follows from the invariant with c2=L25 (process 2 has terminated) and $0 \leqslant n \leqslant 1$, we will also show that the value of N after the parallel command is either 0 or 1. Since the initial value $\underline{n}$ of N is assumed to be positive, the only difficulty is for $\underline{n} > 1$. In this case N is decremented until reaching value 2. On one hand both processes can test that N>1 before it is decremented by the other one, then each process will decrement N and terminate. In this case N would equal 0 on exit of the parallel command. On the other hand, when N=2, one process can test for N>1 and decrement N to 1 before the other process tests for N>1. Then both processes terminate and N=1 on exit of the parallel command. For an invariance proof, the above operational arguments can be rephrased in a "time independent manner", which leads to the following local invariants:

$$Q_{11}(\underline{n},c2,n,p1,p2) = Q_{12}(\underline{n},c2,n,1,p2)$$

$$Q_{12}(\underline{n},c2,n,p1,p2) = [Inv(\underline{n},c2,n,p1,p2) \wedge [(c2 \in \{L21,L22\} \wedge n=\underline{n} \wedge n \geqslant 0) \vee (c2=L23 \wedge n>1) \vee (c2=L24 \wedge n \geqslant 1) \vee (c2=L25 \wedge 0 \leqslant n \leqslant 1)]]$$

$$Q_{13}(\underline{n},c2,n,p1,p2) = [Inv(\underline{n},c2,n,p1,p2) \wedge [(c2 \in \{L21,L22,L23\} \wedge n>1) \vee (c2=L24 \wedge n \geqslant 1) \vee (c2=L25 \wedge n=1)]]$$

$$Q_{14}(\underline{n},c2,n,p1,p2) = [Inv(\underline{n},c2,n,p1,p2) \wedge [(c2 \in \{L21,L22,L23\} \wedge n>0) \vee (c2=L24 \wedge n \geqslant 0) \vee (c2=L25 \wedge 0 \leqslant n \leqslant 1)]]$$

$$Q_{15}(\underline{n},c2,n,p1,p2) = [Inv(\underline{n},c2,n,p2,p1) \wedge [(c2=L23 \wedge n=1) \vee (c2 \neq L23 \wedge 0 \leqslant n \leqslant 1)]$$

$$Q_1(\underline{n},n,p1,p2) = [p1 \times p2 = 2^{\underline{n}-n} \wedge 0 \leqslant n \leqslant 1]$$

$$Q_2(\underline{n},p) = [p=2^{\underline{n}}]$$

It is a simple mathematical exercise to show that these local invariants satisfy the following verification conditions (which are universally quantified over $\underline{n},n,n',p1,p1',p2,p2',p \in \Pi$, $c1 \in \{L11,...,L15\}$):

– Initialization:

$$[\underline{n} \geqslant 0] \Rightarrow [Q_{11}(\underline{n},L21,\underline{n},p1,p2) \wedge Q_{21}(\underline{n},L11,\underline{n},p2,p1)]$$

– Sequential proof (similar to E.1):

$$[Q_{11}(\underline{n},c2,n,p1',p2) \wedge 1 \in \Pi] \Rightarrow Q_{12}(\underline{n},c2,n,1,p2)$$

$$[Q_{12}(\underline{n},c2,n,p1,p2) \wedge n>1] \Rightarrow Q_{13}(\underline{n},c2,n,p1,p2)$$

$$[Q_{12}(\underline{n},c2,n,p1,p2) \wedge n \leqslant 1] \Rightarrow Q_{15}(\underline{n},c2,n,p1,p2)$$

$$[Q_{13}(\underline{n},c2,n',p1',p2) \wedge (n'-1) \in \Pi \wedge (2 \times p1') \in \Pi] \Rightarrow Q_{14}(\underline{n},c2,n'-1,2 \times p1',p2)$$

$$[Q_{14}(\underline{n},c2,n,p1,p2) \wedge n>1] \Rightarrow Q_{13}(\underline{n},c2,n,p1,p2)$$

$$[Q_{14}(\underline{n},c2,n,p1,p2) \wedge n \leqslant 1] \Rightarrow Q_{15}(\underline{n},c2,n,p1,p2)$$

– Interference freeness check (for $k=1,...,5$):

$$[Q_{1k}(\underline{n},L21,n,p1,p2') \wedge Q_{21}(\underline{n},L1k,n,p2',p1) \wedge 1 \in \Pi] \Rightarrow Q_{1k}(\underline{n},L22,n,p1,1)$$

$$[Q_{1k}(\underline{n},L22,n,p1,p2) \wedge Q_{22}(\underline{n},L1k,n,p2,p1) \wedge n>1] \Rightarrow Q_{1k}(\underline{n},L23,n,p1,p2)$$

$$[Q_{1k}(\underline{n},L22,n,p1,p2) \wedge Q_{22}(\underline{n},L1k,n,p2,p1) \wedge n \leqslant 1] \Rightarrow Q_{1k}(\underline{n},L25,n,p1,p2)$$

$$[Q_{1k}(\underline{n},L23,n',p1,p2') \wedge Q_{23}(\underline{n},L1k,n',p2',p1) \wedge (n'-1) \in \Pi \wedge (2 \times p2') \in \Pi] \Rightarrow Q_{1k}(\underline{n},L24,n'-1,p1,2 \times p2')$$

$$[Q_{1k}(\underline{n},L24,n,p1,p2) \wedge Q_{24}(\underline{n},L1k,n,p2,p1) \wedge n>1] \Rightarrow Q_{1k}(\underline{n},L23,n,p1,p2)$$

$$[Q_{1k}(\underline{n},L24,n,p1,p2) \wedge Q_{24}(\underline{n},L1k,n,p2,p1) \wedge n \leqslant 1] \Rightarrow Q_{1k}(\underline{n},L25,n,p1,p2)$$

– Finalization:

$[Q_{15}(\underline{n},L25,n,p1,p2) \wedge Q_{25}(\underline{n},L15,n,p2,p1)] \Rightarrow Q_1(\underline{n},n,p1,p2)$

$[Q_1(\underline{n},n,p1,p2) \wedge n=0 \wedge (p1 \times p2) \in \Pi] \Rightarrow Q_2(\underline{n},p1 \times p2)$

$[Q_1(\underline{n},n,p1,p2) \wedge n \neq 0 \wedge (2 \times p1 \times p2) \in \Pi] \Rightarrow Q_2(\underline{n},2 \times p1 \times p2)$

$Q_2(\underline{n},p) \Rightarrow [p=2^{\underline{n}}]$.

Notice that for the whole program we have got 77 verification conditions. Theoretically the number of sequential verification conditions is linear in the size of the program whereas the number of verification conditions for checking interference freeness grows exponentially with the number and size of processes. The practical method for avoiding this combinatorial explosion is to make informal proofs and to choose the local invariants of each process as independent as possible of the other processes. In that case most of the interference freeness checks become trivial.

F.2 Decomposition of the Global Program Invariant Leading to Lamport [1980] Proof Method

F.2.1 Decomposition

Another way to avoid the proliferation of simple verification conditions is to use a coarser decomposition which consists in associating a global invariant Q_i with each process P_i of program $[\![P_1 |..| P_k]\!]$. Each predicate Q_i may depend upon the values of variables as well as upon program control locations. The correspondence with induction principle D2 is established along the lines of paragraph E by defining global inductive invariant I as the conjunction of the global invariants Q_i for each process P_i:

$$I = \rho'(Q) = \bigwedge_{i=1}^{k} Q_i$$

F.2.2 Derivation of the verification conditions

For the basis D.2(a) we must prove that I is initially true and this verification condition can be decomposed into checks that each $Q_i, i = 1,...,k$ is initially true.

The induction step D.2(b) which consists in proving that I is invariant, that is $I \wedge t \Rightarrow I$, can be decomposed into proofs that $I \wedge t \Rightarrow Q_i$ for $i=1,...,k$. Moreover, the induction hypothesis $I = \bigwedge_{j=1}^{k} Q_j$ can be weakened and one can choose (simpler but sufficient since stronger) conditions $(\bigwedge_{j \in S_i} Q_j) \wedge t \Rightarrow Q_i$ for $S_i \subset \{1,...k\}$ and $S_i \neq \emptyset$. (These verification conditions satisfy the completeness criterion of paragraph E for $\rho(I) = Q$ iff $Q_i = I$ for $i = 1,...k$. This formalizes the intuitive idea that the method is complete since all global invariants Q_i for each process $i=1, ...,k$ can always be chosen as the global program invariant which is used for the completeness proof of induction principle D.2.) A further decomposition of the verification conditions $(\bigwedge_{j \in S_i} Q_j) \wedge t \Rightarrow Q_i$ is possible since t is a disjunction of cases. For each process i, one can distinguish between a sequential proof $(\bigwedge_{j \in S_{il}} Q_j) \wedge t_{il} \Rightarrow Q_i$ (when t corresponds to execution of a basic command

labelled l of process i) and an interference freeness check $(\bigwedge_{j \in S_{hl}} Q_j) \wedge t_{hl} \Rightarrow Q_i$ (when t corresponds to execution of a basic command labelled l of process $h \neq i$).

F.2.3 Example

Let us give another proof of program B.2, using process invariants. Obviously, the proof is just a reformulation of F.1.3 using a global invariant for each process instead of local invariants attached to program points. Since both processes are symmetric, we use the same process invariant for each of them and reason only on process one.

In order to be able to designate program locations let us introduce:

$$\begin{aligned}
\text{Not-started} &\equiv (c1 = L11 \wedge c2 = L21)\\
\text{P2-started} &\equiv (c1 = L11 \wedge c2 \neq L21)\\
\text{P1-started} &\equiv (c1 \neq L11 \wedge c2 = L21)\\
\text{Started} &\equiv (c1 \neq L11 \wedge c2 \neq L21)
\end{aligned}$$

The central idea of program B.2 is to maintain invariant the following relation:

$$\begin{aligned}
\text{Inv}(\underline{n},c1,c2,n,p1,p2) = [&(\text{Not-started} \wedge n = \underline{n}) \vee (\text{P2-started} \wedge p2 = 2^{\underline{n}-n})\\
&\vee (\text{P1-started} \wedge p1 = 2^{\underline{n}-n}) \vee (\text{Started} \wedge p1 \times p2 = 2^{\underline{n}-n}).
\end{aligned}$$

The other essential observation for the partial correctness proof is that the program can only terminate when $0 \leqslant N \leqslant 1$. To prove this, let us introduce:

$$\begin{aligned}
&\text{Before-test} \equiv [(c1 \in \{L11, L21\} \wedge c2 \in \{L21, L22\}) \vee (c1 = L14 \wedge c2 = L24)]\\
&\text{After-test} \equiv [(c1 = L13 \wedge c2 \in \{L21, L22, L23\}) \vee (c1 \in \{L11, L12, L13\} \wedge c2 = L23)]\\
&\text{After-test-and-decrement} \equiv [(c1 = L14 \wedge c2 \in \{L21, L22, L23\})\\
&\qquad\qquad \vee (c1 \in \{L11, L12, L13\} \wedge c2 = L24)]\\
&\text{One-decrement-left} \equiv [(c1 = L15 \wedge c2 = L23) \vee (c1 = L13 \wedge c2 = L25)]\\
&\text{No-decrement-left} \equiv [(c1 = L15 \wedge c2 \neq L23) \vee (c1 \neq L23 \wedge c2 = L25)]
\end{aligned}$$

For each process, one can choose the following global invariant:

$$\begin{aligned}
Q(\underline{n},c1,c2,n,p1,p2) = [&\text{Inv}(\underline{n},c1,c2,n,p1,p2) \wedge [(\text{Before-test} \wedge n \geqslant 0)\\
&\vee (\text{After-test} \wedge n > 1) \vee (\text{After-test-and-decrement} \wedge n \geqslant 1)\\
&\vee (\text{One-decrement-left} \wedge n = 1) \vee (\text{No-decrement-left} \wedge 0 \leqslant n \leqslant 1)]]
\end{aligned}$$

This global invariant satisfies the following verification conditions:

– Initialization.

$$[\underline{n} \geqslant 0] \Rightarrow Q(\underline{n},L11,L21,\underline{n},p1,p2)$$

– Sequential proof of process 1:

$$[Q(\underline{n},L11,c2,n,p1',p2) \wedge 1 \in \Pi] \Rightarrow Q(\underline{n},L12,c2,n,\ 1,p2)$$

$$[Q(\underline{n},L12,c2,n,p1,p2) \wedge n>1] \Rightarrow Q(\underline{n},L13,c2,n,p1,p2)$$

$$[Q(\underline{n},L12,c2,n,p1,p2) \wedge n \leqslant 1] \Rightarrow Q(\underline{n},L15,c2,n,p1,p2)$$

$$[Q(\underline{n},L13,c2,n',p1',p2) \wedge (n'-1) \in \Pi \wedge (2 \times p1') \in \Pi] \Rightarrow Q(\underline{n},L14,c2,n'-1,2 \times p1',p2)$$

$$[Q(\underline{n},L14,c2,n,p1,p2) \wedge n>1] \Rightarrow Q(\underline{n},L13,c2,n,p1,p2)$$

$$[Q(\underline{n},L14,c2,n,p1,p2) \wedge n \leqslant 1] \Rightarrow Q(\underline{n},L15,c2,n,p1,p2)$$

– The proof of absence of interference of execution of process 2 with the global invariant of process 1 exactly amounts to the sequential proof of process 2.

– Finalization:

$$Q(\underline{n},L15,L25,n,p1,p2) \Rightarrow Q_1(\underline{n},n,p1,p2)$$

When compared with F.1.3 the use of a coarser decomposition leads, for that example, to a natural factorization of similar verification conditions.

F.3 Classification of Program Proof Methods

Program proof methods can be classified according to the class of properties that can be proved. Methods for proving properties in a given class can be classified according to the basic induction principle underlying them. Finally for a given induction principle, proof methods can be compared according to the decomposition of the global inductive hypothesis involved in this induction principle into a set of local inductive hypotheses.

Decompositions can be partially ordered, a decomposition being coarser than another if the former can be further decomposed into the latter.

Example F.3.1 A comparison of two invariance proof methods.

The decomposition of the global program invariant of D2 into the global process invariants of F.2 is coarser than the decomposition into the local invariants of F.1. If we have proved program $[\![P_1 \,|.|\, P_k]\!]$ using global process invariants $G_1,...,G_k$ we can rephrase the proof using local invariants Q_{il}, $i \in \{1,...,k\}$, $l \in C_i$ which are the decomposition of the G_i, in the sense of paragraph E, that is:

$$Q = \rho(G) \quad \text{s.t.} \quad Q_{il}(c_1,...,c_{i-1},c_{i+1},...,c_k,x) = G_i(c_1,...,c_{i-1},l,c_{i+1},...,c_k,x).$$

Rcciprocally, a proof using global process invariants can be derived from a proof using local invariants using

$$G = \rho'(Q) \text{ s.t. } G_i(c_1,...,c_k,x) = \bigvee_{l \in C_i} [(c_i=l) \wedge Q_{il}(c_1,...,c_{i-1},c_{i+1},...,c_k,x)]$$

since ρ is bijective and ρ' its inverse the methods are equivalent. □

The above informal idea of comparing proof methods corresponding to a given induction principle according to the decomposition of a global invariant into local invariants can be made formal. The partial ordering on program proof methods is chosen as the ordering on the closure operators $\rho' o \rho$ on the partially ordered set of global invariants $(A, =>)$, induced by the Galois connections (ρ, ρ') corresponding to each of these methods. In this way, a complete lattice of proof methods is obtained, which is part of the lattice of program analysis methods considered in Cousot and Cousot [1979].

G. Flow Analysis of Parallel Programs

Approximate semantic analysis of programs also called program flow analysis (Muchnick and Jones [1981]) is "a tool for discovering properties of the run-time behavior of a program without actually running it. The properties discovered usually apply to all possible sequences of control and data flow and so give global information impossible to obtain by individual runs or by inspection of only a part of the program."

G.1 Design of a Flow Analysis Algorithm

Almost all program flow analysis techniques have been designed for sequential programs. Using the approach of Cousot and Cousot [1979], Cousot, P. [1981], the extension of these techniques to parallel programs is trivial. It consists in choosing an induction principle and a decomposition (α, γ) of the global invariant into a set of machine representable local invariants. Once the corresponding verification conditions have been designed, they are considered as a fixed point system of equations $F(X) => X$ (or $X => F(X)$), the least (or greatest) fixed point D of which leads to the best (i.e. more precise) possible set of local invariants (for the given decomposition (α, γ)) since $F(D) = D$ and if $F(X) => X$ then $D => X$ (or $X => F(X)$ then $X => D$). In order to avoid computability problems, the decomposition is chosen to be approximate, so that the verification conditions F are incomplete. If the approximation is strong enough the least (greatest) fixed point of F can be machine computed as $\bigvee F^i(\bot)$ (or $\bigwedge F^i(\top)$) where $\bot$ is $\rho(\text{ff})$ (resp. $\rho(\text{tt})$) and $F^o(X) = X$, $F^{i+1}(X) = F(F^i(X))$. If the convergence is not rapid enough extrapolation techniques are available (Cousot and Cousot [1977]).

G.2 Example of Approximate Decomposition for Parallel Programs

In order to illustrate how our approach to program flow analysis applies equally well to parallel programs than to sequential programs, we will use program B.2 (the reasoning on this program can be extended to a programming language via induction on the syntax of programs).

Assume we want to determine statically for each possible point of control in the program a finite description of the set of data states the program could be in when execution passes through that point. The description D_{jl} attached to point l of process $j=1,2$ is chosen to be an interval of values for each variable and each point $h \in C_{\tilde{j}}$ of the other process $\tilde{j}$ (where $\tilde{1}=2 \wedge \tilde{2}=1$). Therefore $D_{jl}(\tilde{j}h)$ is a triple $<n,p_j,p_{\tilde{j}}>$ of abstract values of variables N, P_j, $P_{\tilde{j}}$ where each $n,p_j,p_{\tilde{j}}$ is either $\bot$ (bottom, which stands for the predicate ff) or a numerical interval [a,b] such that $li \leqslant a \leqslant b \leqslant hi$ where li and hi are the lowest and greatest machine

representable integer. We will use the selectors $<n,p_j,p_{\tilde{j}}>[N]=n$, $<n,p_j,p_{\tilde{j}}>[P_i]=p_j$ and $<n,p_j,p_{\tilde{j}}>[P_{\tilde{j}}]=p_{\tilde{j}}$. The meaning of a description D can be explained in terms of the local invariants Q considered at paragraph F.1 by the connection $Q=\gamma(D)$ such that:

$$Q_{j\tilde{l}}(\underline{n},c_{\tilde{j}},n,p_j,p_{\tilde{j}}) = \bigvee_{h\in C_{\tilde{j}}} ((c_{\tilde{j}}=h) \wedge (\forall v\in\{N,P1,P2\},\ D_{j\tilde{l}}(\tilde{jh})[v]\neq\perp \wedge v\in D_{j\tilde{l}}(\tilde{jh})[v])).$$

For example $D_{14}(23) = <[1,hi-1],[2,hi],[1,hi]>$ means that at point L14 of process 1 it is true that $((1\leqslant n\leqslant hi-1)\wedge(2\leqslant p_1\leqslant hi)\wedge(1\leqslant p_2\leqslant hi))$ when control is at point L23 of process 2. Reciprocally a set of local invariants Q can be approximated by $D=\alpha(Q)$ such that

$$D_{j\tilde{l}}(\tilde{jh}) = <\perp,\perp,\perp> \text{ iff } Q_{j\tilde{l}}(\underline{n},L_{\tilde{jh}},n,p_j,p_{\tilde{j}}) = ff$$

$$= <[\min V(N),\max V(N)],[\min V(P_j),\max V(P_j)],[\min V(P_{\tilde{j}}),\max V(P_{\tilde{j}})]>$$

where

$$V(N) = \{n\in\Pi \mid (\exists\, \underline{n},p_j,p_{\tilde{j}}\in\Pi \mid Q_{j\tilde{l}}(\underline{n},L_{\tilde{jh}},n,p_j,p_{\tilde{j}})\}$$

and similarly for $V(P_j),V(P_{\tilde{j}})$.

Notice that $\alpha(Q)$ is an approximation of Q in that, for example, relationships between variables cannot be expressed.

G.3 Fixed Point System of Approximate Equations Associated with a Parallel Program

The verification conditions of paragraph F.1.3 can be written as a system of inequations $Q<=V(Q)$ which was obtained by decomposition of

$$I <= \lambda(\underline{s},s)\ .\ [\epsilon(\underline{s}) \vee (\exists s' \mid I(\underline{s},s') \wedge t(s',s))]$$

using the connection (ρ,ρ') defined at paragraph F.1.1. A further decomposition leads to a fixed point system of approximate equations $D=F(D)$ using the connection (α,γ) of paragraph G.2 and $F=\alpha o V o \gamma$. Then the meaning of the least fixed point lfp(F) of F is an invariant of the program, that is $\forall \underline{s},s\in S,\ [\epsilon(\underline{s})\wedge t^*(\underline{s},s)] => \rho' o \alpha(\text{lfp}(F))(s)$.

Before giving the system of equations corresponding to program B.2 let us introduce some notations:

- $<n,p1,p2>[p1:p1'] = <n,p1',p2>$ — substitution
  $<n,p1,p2>[p2:p2'] = <n,p1,p2'>$

- $\perp\wedge x=x\wedge\perp=\perp$ for $x\in\{\perp\}\cup\{[a,b]\mid a\leqslant b\}$ — approximate conjunction
  $[a,b]\wedge[c,d]=[\max(a,c),\min(b,d)]$ iff $\max(a,c)\leqslant\min(b,d)$
  $=\perp$ iff $b<c$ or $d<a$

- $\perp\vee x=x\vee\perp=x$ for $x\in\{\perp\}\cup\{[a,b]\mid a\leqslant b\}$ — approximate disjunction
  $[a,b]\vee[c,d]=[\min(a,c),\max(b,d)]$

$-\perp - 1 = \perp$ approximate decrement

$[a,b] - 1 = [a-1,b-1] \wedge [li,hi]$

$-2\times\perp = \perp$ approximate shift.

$2\times[a,b] = [2\times a,2\times b] \wedge [li,hi]$

In the following presentation of the fixed point system of approximate equations for program B.2, it is assumed that initially we must have $n \geqslant 0$. For each equation we distinguish a term corresponding to the sequential proof and a term corresponding to the interference check:

$$D_0 = <[0,hi],[li,hi],[li,hi]>$$

$$D_{11}(21) = D_0$$

$$D_{11}(2k) = inter_{11}(2k) \qquad k = 2,...,5$$

$$D_{12}(2k) = D_{11}(2k)[p1:[1,1]] \vee inter_{12}(2k) \qquad k = 1,...,5$$

$$D_{13}(2k) = (D_{12}(2k) \wedge <[2,hi],[li,hi],[li,hi]>) \vee (D_{14}(2k) \wedge <[2,hi],[li,hi],[li,hi]>) \vee inter_{13}(2k) \qquad k = 1,...,5$$

$$D_{14}(2k) = <D_{13}(2k)(n)-1,\ 2\times D_{13}(2k)(p1),\ D_{13}(2k)(p2)> \vee inter_{14}(2k)$$

$$D_{15}(2k) = (D_{12}(2k) \wedge <[li,1],[li,hi],[li,hi]>) \vee (D_{14}(2k) \wedge <[li,1],[li,hi],[li,hi]>) \vee inter_{14}(2k) \qquad k = 1,...,5$$

where

$$inter_{1k}(21) = <\perp,\perp,\perp>$$

$$inter_{1k}(22) = (D_{1k}(21) \wedge D_{21}(1k))[p2:[1,1]]$$

$$inter_{1k}(23) = (D_{1k}(22) \wedge D_{22}(1k) \wedge <[2,hi],[li,hi],[li,hi]>) \vee (D_{1k}(24) \wedge D_{24}(1k) \wedge <[2,hi],[li,hi],[li,hi]>)$$

$$inter_{1k}(24) = <(D_{1k}(23)(n) \wedge D_{23}(1k)(n))-1,\ D_{1k}(23)(p1) \wedge D_{23}(1k)(p1),\ 2\times(D_{1k}(23)(p2) \wedge D_{23}(1k)(p2))>$$

$$inter_{1k}(25) = (D_{1k}(22) \wedge D_{22}(1k) \wedge <[li,1],[li,hi],[li,hi]>) \vee (D_{1k}(24) \wedge D_{24}(1k) \wedge <[li,1],[li,hi],[li,hi]>)$$

... similar equations for process 2 ....

$$D_1 = D_{15}(25) \wedge D_{25}(15)$$

G.4 Iterative Resolution of the Equations Using Extrapolation Techniques for Accelerating the Convergence

This system of equations can be solved using any asynchronous iterative strategy (Cousot, P. [1977]). Initially one set $D_{il}(\tilde{ih}) = <\perp,\perp,\perp>$ for $i = 1,2$, $l \in C_i$, $h \in C_{\tilde{i}}$. Then one iterates through the system applying any equation until no changes take place.

The convergence can be accelerated using Cousot and Cousot [1976,1977] extrapolation techniques. This consists in defining a widening operation ∇ such as:

$$\bot \nabla x = x$$

$$[a,b] \nabla [c,d] = [\ \underline{\text{if}}\ c < a\ \underline{\text{then}}\ li\ \underline{\text{else}}\ a,\ \underline{\text{if}}\ d > b\ \underline{\text{then}}\ hi\ \underline{\text{else}}\ b]$$

and replacing equations $D_{j3}{}_j = 1,2$ by

$$D_{j3}(\tilde{jk}) = D_{j3}(\tilde{jk}) \nabla [(D_{j2}(\tilde{jk}) \wedge <[2,hi],[li,hi],[li,hi]>) \vee (D_{j4}(\tilde{jk}) \wedge <[2,hi],[li,hi],[li,hi]>) \vee inter_{j3}(\tilde{jk})]$$

and then solving iteratively. The result we have obtained (for process 1) is:

	k=1			k=2			k=3			k=4			k=5		
	n	p1	p2	n	p1	p2	n	p1	p2	n	p1	p2	n	p1	p2
$D_{11}(2k)$	[0,hi]	[li,hi]	[li,hi]	[0,hi]	[li,hi]	[1,1]	[2,hi]	[li,hi]	[1,hi]	[1,hi-1]	[li,hi]	[2,hi]	[0,1]	[li,hi]	[1,hi]
$D_{12}(2k)$	[0,hi]	[1,1]	[li,hi]	[0,hi]	[1,1]	[1,1]	[2,hi]	[1,1]	[1,hi]	[1,hi-1]	[1,1]	[2,hi]	[0,1]	[1,1]	[1,hi]
$D_{13}(2k)$	[2,hi]	[1,hi]	[li,hi]	[2,hi]	[1,hi]	[1,1]	[2,hi]	[1,hi]	[1,hi]	[1,hi-1]	[1,hi]	[2,hi]	[1,1]	[1,hi]	[1,hi]
$D_{14}(2k)$	[1,hi-1]	[2,hi]	[li,hi]	[1,hi-1]	[2,hi]	[1,1]	[1,hi-1]	[2,hi]	[1,hi]	[0,hi-2]	[2,hi]	[2,hi]	[0,1]	[2,hi]	[1,hi]
$D_{15}(2k)$	[0,1]	[1,hi]	[li,hi]	[0,1]	[1,hi]	[1,1]	[1,1]	[1,hi]	[1,hi]	[0,1]	[1,hi]	[2,hi]	[0,1]	[1,hi]	[1,hi]

Notice that the decomposition is approximate enough to allow a computer implementation of this kind of analyses. However, as shown by the above example, the results of such approximate analyses can be useful since we obtain

$$D_1 = <[0,1],[1,hi],[1,hi]>$$

which proves that $N \in \{0,1\}$ on exit of the parallel command of program B.2, a result which is not trivial to obtain by hand.

References

Cousot P. [1977]
P. Cousot, "Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice," Research Report No. 88, IMAG, University of Grenoble, France (Sept. 1977).

Cousot P. [1978]
P. Cousot, "Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes," Thése d'Etat, University of Grenoble, France (March 1978).

Cousot P. [1979]
P. Cousot, "Analysis of the behavior of dynamic discrete systems," Research Report No. 161, IMAG, University of Grenoble, France (Jan. 1979).

Cousot P. [1981]
P. Cousot, "Semantic foundations of program analysis," in *Program Flow Analysis, Theory and Applications,* S.S Muchnick & N.J. Jones (eds.), Prentice-Hall, Inc. (1981), pp. 303−342.

Cousot R. [1981]
R. Cousot, "Proving invariance properties of parallel programs by backward induction," Research Report, CRIN−81−P026, Nancy, France (March 1981), to appear in *Acta Informatica.*

Cousot and Cousot [1976]
P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," *Proc. 2nd Int. Symp. on Programming,* Dunod, Paris, France (April 1976), pp. 106−130.

Cousot and Cousot [1977]
P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," *Conf. Rec. of 4th ACM Symp. on Principles of Programming Languages,* Los Angeles, CA. (Jan. 1977), pp. 238−252.

Cousot and Cousot [1979]
P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," *Conf. Rec. of 6th ACM Symp. on Principles of Programming Languages,* San Antonio, TX (Jan. 1979), pp. 269−282.

Cousot and Cousot [1980a]
P. Cousot and R. Cousot, "Semantic analysis of communicating sequential processes," *Automata, Languages and Programming, 7th Colloq. Noordwijkerhout, Lecture Notes in Comp. Science 85,* Springer-Verlag (July 1980), pp. 119−133.

Cousot and Cousot [1980b]
P. Cousot and R. Cousot, "Constructing program invariance proof methods," *Proc. Int. Workshop on Program Construction,* Chateau de Bonas, France, Tome 1 INRIA Ed. (Sept. 1980).

Cousot and Cousot [1982]
P. Cousot and R. Cousot, "Induction principles for proving invariance properties of programs," in *Tools and Notions for Program Construction,* Nice, France, (Dec. 7-18, 1981), Cambridge University Press (1982), pp. 75−119.

Cousot and Halbwachs [1978]
P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," *Conf. Rec. of 5th ACM Symp. on Principles of Programming Languages,* Tuscon, AZ (Jan. 1978), pp. 84−97.

Floyd [1967]
R.W. Floyd, "Assigning meanings to programs," *Proc. Symp. in Applied Math.,* Vol. 19, AMS, Providence, RI (1967), pp. 19−32.

Hoare [1969]
C.A.R. Hoare, "An axiomatic basis for computer programming," *C.ACM* 12, 10(Oct. 1969), pp. 576−580, 583.

Keller [1976]
R.M. Keller, "Formal verification of parallel programs," *C.ACM* 19, 7(July 1976), pp. 371−384.

Lamport [1977]
L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. on Soft. Eng.*, SE3, 2(March 1977), pp. 125−143.

Lamport [1980]
L. Lamport, "The 'Hoare Logic' of concurrent programs," *Acta Informatica* 14 (1980), pp. 21−37.

Muchnick and Jones [1981]
S.S. Muchnick and N.D. Jones (eds.), *Program Flow Analysis: Theory and Applications,* Prentice-Hall Inc. (1981).

Naur [1966]
P. Naur, "Proof of algorithms by general snapshots," *BIT 6* (1966), pp. 310−316.

Owicki and Gries [1976]
J. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I," *Acta Informatica* 6 (1976), pp. 319−340.