SYNTHESIS OF PARALLEL PROGRAMS INVARIANTS

E. Pascal Gribomont
Institut Montefiore
Université de Liège
4000 Liège, Belgium

ABSTRACT.

Most current methods for parallel programs design and verification are based on the concept of invariant. However, invariant synthesis is the most difficult part of those methods. This work presents a technique for invariant design usable for both parallel program synthesis and verification. This technique drastically reduces the risks of errors due to a bad statement serialization or to inadequate interprocess synchronization.

1. INTRODUCTION.

The verification and implementation of parallel programs meeting given requirements is difficult. Several methods have been proposed to specify and verify concurrent processes (see e.g. [1] and [11]). In most of these methods, the interesting properties of programs are often expressed as a relation between program variables and control points. Such a relation, named "safety property" or "invariance property", must remain true throughout the execution. To prove these properties, the invariant principle is generally used. Here we name "invariant" a safety property which is inductive, i.e. preserved by each statement of the program. Checking an invariant is straightforward and thus a simple way to prove some safety property is to find an invariant which implies this property. This approach was introduced by Floyd, Hoare and Dijkstra for sequential programming (see [3] for instance) and adapted to parallel programming by many authors e.g. Ashcroft, Owicki, Gries and Levin (see [1], [11] and [9]). Unfortunately, such methods have a major drawback : the design of parallel program invariants is not easy.

The technique we present in this paper is based on a very simple observation : simple programs generally admit simple invariants. Our design method consists in applying successive refinements to both the program and its invariant. The starting point is an abstract and simplified version of the program which usually admits a fairly simple invariant. The goal is of course the real program but also its complete invariant. Our method can also be used for program verification.

This research has been supported in part by IRSIA under grant number 80018.

The expression "refinement" has received many meanings, in the field of computer science (for instance, in the top-down sequential programming methodology). In this work, the two fundamental and opposite operations on a pair (program, invariant), called refinement and abstraction, have a precise meaning.

The main kind of refinement consists in transforming a non-elementary statement (like multiple assignment) into a sequence of simpler ones producing the same global effect. This transformation, called "splitting", does not change anything from a strict sequential point of view but introduces more interleaving between the processes, from the parallel point of view. The invariant must be adapted to take the new control points into account. More precisely, if I is the invariant of the program before splitting and J is the invariant after, we can write $J = I \wedge A$, where A is an unknown formula. This formula is a solution of a set of boolean inequations obtained from the text of the program before and after the transformation. These inequations are of two kinds : sequential constraints are related to the process containing the transformed statement and interaction constraints correspond to the other processes.

Another kind of refinement is the replacement of a high level communication statement by a lower level statement or set of statements. For instance, CSP-like communication statements, which allow only synchronous communication, can be replaced by classical send-receive statements, which allow asynchronous communication. Such transformations give rise to new control points, corresponding to states in which some messages have been sent but not yet received.

This incremental synthesis technique simplifies invariant design. At each step, the user has to perform a simple task : to state a system of boolean inequations and to find a solution of it (if possible). This method has been successfully applied to synthesize an invariant for the "On-the-fly garbage collector" due to Dijkstra (see [8] and [5]). Communicating processes have also been investigated by this technique. The algorithm of Ricart and Agrawala, which provides mutual exclusion in a computer network and was not formally verified (in fact, it contained a bug), has been studied by this method (see [7], [13] and [14]).

In the rest of this paper, the method is developed and applied to an example (a more detailed presentation of the method appears in [6]). While the example is rather short, the corresponding invariant is not trivial (see [4]). It emphasizes the difficulty of avoiding sequencing or synchronization errors in parallel programming.

2. EXAMPLE.

We wish to implement the mutual exclusion between the critical sections of two concurrent processes. More precisely, there are two cyclic processes attempting from time to time to execute a critical section. The implementation must be such that they will never be both in their critical section. The solution presented here has been published with an informal proof by Peterson (see [12]).

2.1. Synthesis of a condensed version.

A starting point can be obtained from two simple classical algorithms, although they both suffer from a critical drawback.

Algorithm 1. (Initial conditions are immaterial).

```
    process P1                          process P2
Repeat forever                      Repeat forever
    Non-critical section                Non-critical section
    T := 2                              T := 1
    T = 1 ?                             T = 2 ?
    Critical section                    Critical section
```

If B is a condition, "B ?" means "wait until B". Variable T may be seen as the "turn". Each process wishing to access its critical section first gives priority to the other process; when it is given priority back, it enters its critical section. This policy involves a difficulty : if one process never wishes to access its critical section, the other will never get the opportunity to enter its own.

Algorithm 2. (Initial conditions : ~Q1 and ~Q2).

```
    process 1                           process 2
Repeat forever                      Repeat forever
    Non-critical section                Non-critical section
    Q1 := true                          Q2 := true
    ~Q2 ?                               ~Q1 ?
    Critical section                    Critical section
    Q1 := false                         Q2 := false
```

Variable Qi (i = 1,2) means "process i requires access to its critical section". There is a problem when both processes simultaneously require access : they will be both deadlocked.Peterson has pointed out that his algorithm can be drawn from these two primitive versions. We will in fact deduce Peterson's algorithm from them.

The two primitive algorithms can be merged. The access scenario to enter the critical section will be :

- The process gives way to the other process and signals its request.
- The access will be granted if the process has the turn or if the other process does not require access.

When the critical section is completed,

The process releases the request.

Two difficulties occur :

- Each process cyclically executes four actions about shared variables Q1, Q2 and T. These actions correspond to four different control points. An invariant taking the sixteen possible pairs into account is needed.
- We do not know if a process requesting access must first give priority and then signal its request or inversely. It is possible that only one of the two serialization proves acceptable.

Without knowing the answer to the last question, we can start with the following program :

```
          (Initial conditions : ~Q1 and ~Q2) :

        process 1                          process 2
    Repeat forever                     Repeat forever
        Non-critical section               Non-critical section
        T,Q1 := 2,true                     T,Q2 := 1, true
        (~Q2 v T=1)?                       (~Q1 v T=2)?
        Critical section                   Critical section
        Q1 := false                        Q2 := false
```

This version is not fine-grained enough for an implementation : there is a multiple assignment. Also each process comprises three control points (see fig. 1); this makes it difficult to find an adequate invariant (if it exists). To find the invariant, we will first consider a condensed version of this algorithm. Afterwards, this version and its invariant will be refined by splitting.

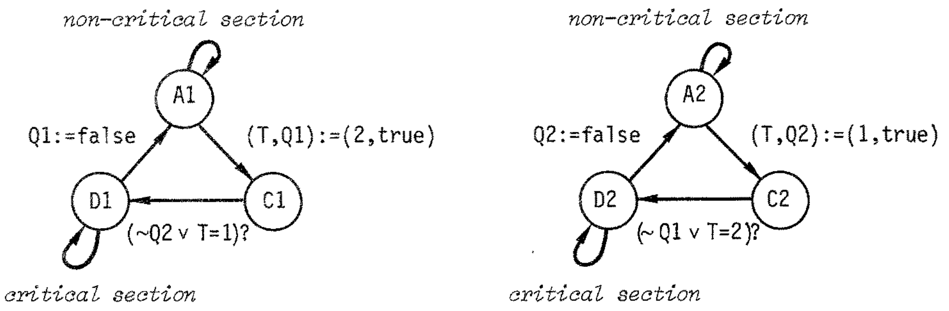

Figure 1.
A first version of the algorithm.

In the condensed version, the double assignment and the test will be abstracted into a single statement.

Let us first introduce some notation. If S is a (sequential) program and P,Q are two predicates, the formula

$$\{P\}\ S\ \{Q\}$$

means that if the execution of S is started in a state satisfying P and if this execution terminates, the final state satisfies Q. Two statements S1 and S2 are equivalent if for all predicates P and Q, we have

$$\{P\}\ S1\ \{Q\} \equiv \{P\}\ S2\ \{Q\}$$

If B is a predicate and S a statement, (B → S) is a statement; it means : "wait until B becomes true and then execute S". If B remains false forever, the execution of (B → S) will never terminate. Formally we can write

$$[\{P \wedge B\}\ (B \rightarrow S)\ \{Q\}] \equiv [\{P \wedge B\}\ S\ \{Q\}]$$

$$[\{P \wedge \sim B\}\ (B \rightarrow S)\ \{Q\}] \equiv \text{true}$$

Such statements are written "Await B then do S" in [11]. Another useful notation is the following. If S is a program, <S> is an atomic statement producing the same net-effect as S. From the sequential point of view, there is no distinction between S and <S>.

For process 1, the abstracted statement

$$<(T,Q1 := 2,\text{true})\ ;\ (\sim Q2 \vee T=1)?\ >$$

is equivalent to

$$< \sim Q2 \rightarrow (T,Q1 := 2,\text{true})>$$

The last statement is

$$< Q1 := \text{false}>$$

The condensed version is depicted on fig. 2. Each process has only two control points. This makes design of the invariant easy :

Mutual exclusion requires :

$$\text{at}(D1,D2) \supset \text{false}$$

Due to initial conditions and exit statements, we have :

$$\text{at}(A1,A2) \supset (\sim Q1 \wedge \sim Q2)$$

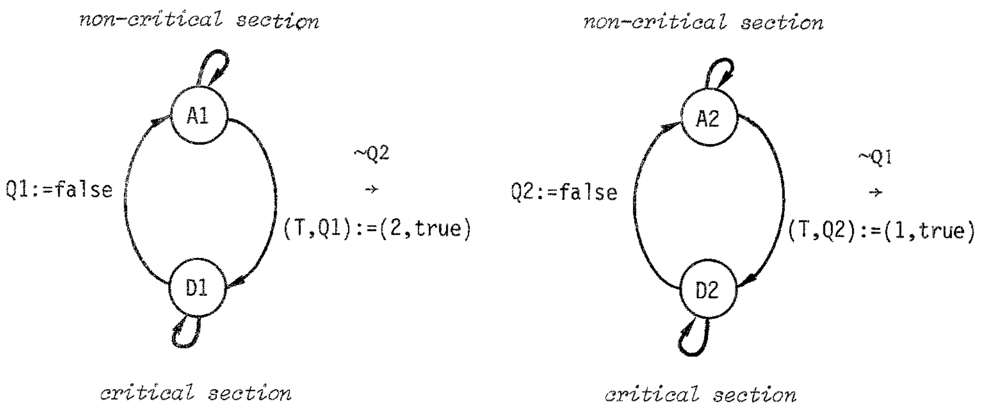

Figure 2.
The condensed version of the algorithm.

The invariant will be completed by two assertions :

$at(A1,D2) \Rightarrow I(A1,D2)$
$at(D1,A2) \Rightarrow I(D1,A2)$

where the formulas $I(A1,D2)$ and $I(D1,A2)$ satisfy the following conditions (we give only those related to $I(D1,A2)$).

$\{I(A1,A2)\} < \sim Q2 \rightarrow (T,Q1 := 2,true) > \{I(D1,A2)\}$
$\{I(D1,A2)\} < \sim Q1 \rightarrow (T,Q2 := 1,true) > \{I(D1,D2)\}$
$\{I(D1,A2)\}\ Q1 := false\ \{I(A1,A2)\}$
$\{I(D1,D2)\}\ Q2 := false\ \{I(D1,A2)\}$

These four constraints correspond to the four transitions starting from or ending to D1 or A2. These conditions form a set of inequations in the boolean lattice $(B, \supset)$ where B is the set of predicates which only admit as free variables the program variables. These conditions amount respectively to the following :

$(Q1 \wedge \sim Q2 \wedge T=2) \supset I(D1,A2)$
$I(D1,A2) \supset Q1$
$I(D1,A2) \supset \sim Q2$
true

$I(D1,A2)$ belongs to the boolean interval defined by :

$$(Q1 \wedge \sim Q2 \wedge T=2) \supset I(D1,A2) \supset (Q1 \wedge \sim Q2)$$

where $p \supset q \supset r$ is an abbreviation for $[p \supset q] \wedge [q \supset r]$. We will adopt here the strongest limit to gain as much knowledge as possible about the program but this is not mandatory. The formula $I(A1,D2)$ is obtained symmetrically; the completed invariant is summarized below.

$at(A1,A2) \supset (\sim Q1 \wedge \sim Q2)$
$at(D1,A2) \supset (Q1 \wedge \sim Q2 \wedge T=2)$
$at(A1,D2) \supset (\sim Q1 \wedge Q2 \wedge T=1)$
$at(D1,D2) \supset \text{false}$

2.2. An intermediate version.

The condensed version and its invariant will be refined in two steps. First, we return to the starting version by splitting the first statement and augmenting the invariant. Next, the multiple assignment will be split and the invariant refined once more.

The splitting of the first statement in process 1 consists in introducing the control point C1 (see fig. 1). Two formulas I(C1,A2) and I(C1,D2) will complete the invariant. We first express the sequential constraints, related to process 1.

- about I(C1,A2) :

$\{I(A1,A2)\}\ T,Q1 := 2,\text{true}\ \{I(C1,A2)\}$
$\{I(C1,A2)\}\ (\sim Q2 \vee T=1)?\ \{I(D1,A2)\}$
which reduces to :
$(Q1 \wedge \sim Q2 \wedge T=2) \supset I(C1,A2) \supset (Q1 \wedge T=2)$

- about I(C1,D2) :

$\{I(A1,D2)\}\ T,Q1 := 2,\text{true}\ \{I(C1,D2)\}$
$\{I(C1,D2)\}\ (\sim Q2 \vee T=1)?\ \{I(D1,D2)\}$
which reduces to :
$(Q1 \wedge Q2 \wedge T=2) \supset I(C1,D2) \supset (Q2 \wedge T=2)$

As before, we prefer the strongest limits of the intervals, with the aim of obtaining as precise an invariant as possible. By the way, let us notice that the following invariance properties are obvious :

$at\ A2 \supset \sim Q2$ $\qquad\qquad$ $at\ D2 \supset Q2$

The interaction constraints, related to process 2, are listed below :

$\{I(C1,A2)\}\ (\sim Q1 \rightarrow T,Q2 := 1,\text{true})\ \{I(C1,D2)\}$
$\{I(C1,D2)\}\ Q2 := \text{false}\ \{I(C1,A2)\}$

They express the fact that the second process respects the new part of the invariant. Checking these constraints is trivial.

The same splitting applies to process 2. Instead of deducing in a similar way the invariant part related to control point C2, we take symmetry into account. This leads to the following formulas :

$I(A1,C2) = (\sim Q1 \wedge Q2 \wedge T=1)$
$I(D1,C2) = (Q1 \wedge Q2 \wedge T=1)$

The assertion corresponding to the state (C1,C2) is determined by four constraints listed below,

$\{I(A1,C2)\}\ T,Q1 := 2,\text{true}\ \{I(C1,C2)\}$
$\{I(C1,C2)\}\ (\sim Q2 \vee T=1)?\ \{I(D1,C2)\}$
$\{I(C1,A2)\}\ T,Q2 := 1,\text{true}\ \{I(C1,C2)\}$
$\{I(C1,C2)\}\ (\sim Q1 \vee T=2)?\ \{I(C1,D2)\}$

which enforce the choice :

$I(C1,C2) = (Q1 \wedge Q2)$

The introduction of control point C2 gives rise to the following interaction constraints :

$\{I(D1,C2)\}\ (D1 \to A1)\ \{I(A1,C2)\}$
$\{I(A1,C2)\}\ (A1 \to C1)\ \{I(C1,C2)\}$
$\{I(C1,C2)\}\ (C1 \to D1)\ \{I(D1,C2)\}$

For instance, we make the first one explicit :

$\{Q1 \wedge Q2 \wedge T=1\}\ Q1 := \text{false}\ \{\sim Q1 \wedge Q2 \wedge T=1\}$

It is trivially true.

COMMENTS : Sequential constraints involve only one unknown formula but interaction constraints involve two of them. That is the reason why the sequential constraints are examined first. They delimit a boolean interval within which the solution must be picked. The choice of a solution satisfying also interaction constraints very often reduces to selecting one of the interval limits (in this example, we always picked the strongest limit). This is also true for more complex programs (examples are presented in [5] and [6]). Nevertheless, this simple tactic sometimes fails : a solution could exist in the interval although the limits do not satisfy the interaction requirements. We met with this situation only once (see [5] for more details). In fact, the method presented here yields inequations which can sometimes admit a number of solutions; the contrary would have been wonderful for a method not restricted to finite state programs.

The complete invariant of the intermediate version is summarized below.

$$at(A1,A2) \supset (\sim Q1 \wedge \sim Q2)$$

$$at(A1,C2) \supset (\sim Q1 \wedge Q2 \wedge T=1) \qquad at(C1,A2) \supset (Q1 \wedge \sim Q2 \wedge T=2)$$

$$at(A1,D2) \supset (\sim Q1 \wedge Q2 \wedge T=1) \qquad at(D1,A2) \supset (Q1 \wedge \sim Q2 \wedge T=2)$$

$$at(C1,C2) \supset (Q1 \wedge Q2)$$

$$at(C1,D2) \supset (Q1 \wedge Q2 \wedge T=2) \qquad at(D1,C2) \supset (Q1 \wedge Q2 \wedge T=1)$$

$$at(D1,D2) \supset false$$

2.3. The final version.

By now, each process comprises three statements, the first of which being a multiple assignment. Our last step will be the splitting of these assignments. The intermediate version invariant will help us to find how these statements can be split.

Let us observe that four cases can occur (while maintaining the symmetry).

1) The intended splitting would endanger mutual exclusion.
2) The splitting is allowed but variable T is to be assigned first.
3) The splitting is allowed but variables Q1, Q2 are to be assigned first.
4) The splitting is allowed and the assignment order is immaterial.

It is well known that intuitive reasoning about parallelism and synchronization is dangerous. The best way is to examine both possible splitting. This is what we will do.

The statement < T,Q1 := 2,true > is split into : T := 2; Q1 := true. The new control point is named B1. This splitting is not depicted on the figures. The sequential constraints are :

- about I(B1,A2) :

  {I(A1,A2)} T := 2 {I(B1,A2)}
  {I(B1,A2)} Q1 := true {I(C1,A2)}
  which reduces to :
  $(\sim Q1 \wedge \sim Q2 \wedge T=2) \supset I(B1,A2) \supset (\sim Q2 \wedge T=2)$

- about I(B1,C2) :

  {I(A1,C2)} T := 2 {I(B1,C2)}
  {I(B1,C2)} Q1 := true {I(C1,C2)}
  which reduces to :
  $(\sim Q1 \wedge Q2 \wedge T=2) \supset I(B1,C2) \supset Q2$

- about I(B1,D2) :

  {I(A1,D2)} T := 2 {I(B1,D2)}
  {I(B1,D2)} Q1 := true {I(C1,D2)}

which reduces to :

$(\sim Q1 \wedge Q2 \wedge T=2) \supset I(B1,D2) \supset (Q2 \wedge T=2)$

The formula $(at\ B1 \supset \sim Q1)$ trivially holds. This leads to the following results :

$at(B1,A2) \supset (\sim Q1 \wedge \sim Q2 \wedge T=2)$

$at(B1,D2) \supset (\sim Q1 \wedge Q2 \wedge T=2)$

About the state (B1,C2), we have two possible assertions :

$at(B1,C2) \supset (\sim Q1 \wedge Q2 \wedge T=2)$

or

$at(B1,C2) \supset (\sim Q1 \wedge Q2)$

The interaction constraint

$\{I(B1,A2)\}\ (A2 \rightarrow C2)\ \{I(B1,C2)\}$

reduces to

$\{\sim Q1 \wedge \sim Q2 \wedge T=2\}\ T,Q2 := 1,\text{true}\ \{\sim Q1 \wedge Q2 \wedge T=2\}$

This eliminates the first possibility. The second one must also be rejected, due to

$\{I(B1,C2)\}\ (C2 \rightarrow D2)\ \{I(B1,D2)\}$

The intended splitting is thus impossible. This failure immediately provides a counterexample of bad execution, which is listed below.

control	Q1	Q2	T
(A1,A2)	F	F	-
(B1,A2)	F	F	2
(B1,C2)	F	T	1
(B1,D2)	F	T	1
(C1,D2)	T	T	1
(D1,D2)	T	T	1

The reverse splitting gives rise to the following statements (see fig. 3) :
(A1, Q1 := true ,B1) and (B1, T := 2 ,C1).

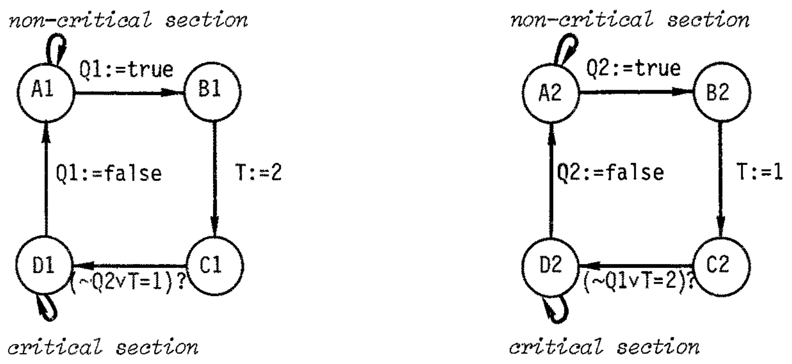

Figure 3.
The final version of the program.

We state now the sequential constraints related to node B1.

- about I(B1,A2) :

  {I(A1,A2)} Q1 := true {I(B1,A2)}
  {I(B1,A2)} T := 2 {I(C1,A2)}
  which reduces to :
  $(Q1 \wedge \sim Q2) \supset I(B1,A2) \supset (Q1 \wedge \sim Q2)$

- about I(B1,C2) :

  {I(A1,C2)} Q1 := true {I(B1,C2)}
  {I(B1,C2)} T := 2 {I(C1,C2)}
  which reduces to :
  $(Q1 \wedge Q2 \wedge T=1) \supset I(B1,C2) \supset (Q1 \wedge Q2)$

- about I(B1,D2) :

  {I(A1,D2)} Q1 := true {I(B1,D2)}
  {I(B1,D2)} T := 2 {I(C1,D2)}
  which reduces to :
  $(Q1 \wedge Q2 \wedge T=1) \supset I(B1,D2) \supset (Q1 \wedge Q2)$

Subject to verifying the interaction constraints, we choose the strongest limits of the three intervals. Here is the list of the interaction constraints, related to process 2.

  {I(B1,A2)} T,Q2 := 1,true {I(B1,C2)}
  {I(B1,C2)} $(\sim Q1 \vee T=2)$? {I(B1,D2)}
  {I(B1,D2)} Q2 := false {I(B1,A2)}

It is easy to check that our choice is valid.

Control point B2 is introduced in the same way. The symmetry leads to the following choices :

$I(A1,B2) = (\sim Q1 \wedge Q2)$
$I(C1,B2) = (Q1 \wedge Q2 \wedge T=2)$
$I(D1,B2) = (Q1 \wedge Q2 \wedge T=2)$

We could compute $I(B1,B2)$ from the usual set of constraints but it is faster to observe that the formulas $(\text{at } B1 \supset Q1)$ and $(\text{at } B2 \supset Q2)$ obviously hold. The symmetry prevents us from fixing the value of T. This leads naturally to :

$I(B1,B2) = (Q1 \wedge Q2)$

Here is the complete invariant of the final version :

$$\begin{array}{lll}
 & at(A1,A2) \supset (\sim Q1 \wedge \sim Q2) & \\
at(A1,B2) \supset (\sim Q1 \wedge Q2) & & at(B1,A2) \supset (Q1 \wedge \sim Q2\ ) \\
at(A1,C2) \supset (\sim Q1 \wedge Q2 \wedge T=1) & & at(C1,A2) \supset (Q1 \wedge \sim Q2 \wedge T=2) \\
at(A1,D2) \supset (\sim Q1 \wedge Q2 \wedge T=1) & & at(D1,A2) \supset (Q1 \wedge \sim Q2 \wedge T=2) \\
 & at(B1,B2) \supset (Q1 \wedge Q2) & \\
at(B1,C2) \supset (Q1 \wedge Q2 \wedge T=1) & & at(C1,B2) \supset (Q1 \wedge Q2 \wedge T=2) \\
at(B1,D2) \supset (Q1 \wedge Q2 \wedge T=1) & & at(D1,B2) \supset (Q1 \wedge Q2 \wedge T=2) \\
 & at(C1,C2) \supset (Q1 \wedge Q2) & \\
at(C1,D2) \supset (Q1 \wedge Q2 \wedge T=2) & & at(D1,C2) \supset (Q1 \wedge Q2 \wedge T=1) \\
 & at(D1,D2) \supset false &
\end{array}$$

3. SOME PROGRAM PROPERTIES.

The mutual exclusion is an immediate consequence of :

$at(D1,D2) \supset false$

This safety property is implied by the invariant. The primitive algorithms both suffer from deadlock; this is not the case for the final version. A deadlock could occur only if both processes were locked, necessarily at C1 and C2 respectively. The invariant implies :

$at(C1,C2) \supset (Q1 \wedge Q2)$

If $T = 1$, process 1 will go on, else process 2 will go on; deadlock is therefore impossible.

Individual starvation is also impossible. Although this fact is not a safety property, it can be deduced from the invariant. Let us suppose, for instance, that process 1 is locked, necessarily at C1. This implies $(Q2 \wedge T = 2)$. On an other hand, the invariant implies (at C1 $\supset$ Q1). This leads to the "lock formula" of process 1 :

$$(Q1 \wedge Q2 \wedge T = 2)$$

Process 2 is not locked and will eventually set T to 1 or Q2 to false; afterwards, process 2 will never gain access until process 1 has executed its critical section.

COMMENTS : The invariant size seems to grow as the product of the processes sizes, which could be unacceptable for large programs. In fact, if suitable notations are used, the invariant size is a good measure of the program complexity : whenever a program can be written, its invariant can also be stated. For medium or large programs, abbreviation techniques are needed. These techniques do not solve by themselves the problem of "combinatorial explosion" but render it less critical. See [6] and [7] for some examples. In our case, the invariant can be shortened as follows :

$$\begin{array}{l} (\text{at } A1 \equiv \sim Q1) \wedge (\text{at } A2 \equiv \sim Q2) \\ (\text{at } C1 \vee \text{at } D1) \supset (T=2 \vee \text{at } C2) \\ (\text{at } C2 \vee \text{at } D2) \supset (T=1 \vee \text{at } C1) \end{array}$$

COMMENT : Our method, like other invariance methods, is especially devoted to invariance properties. Such properties assert that something bad never occurs (falsification of an invariant, deadlock, termination with incorrect results,...). Nevertheless, liveness properties, which assert that something good will eventually happen, can be established easier if adequate safety properties have been proved before. Notice that the invariant was needed to establish freeness of starvation. The same is also true in sequential programming : for instance, a proof of total correctness (liveness property) often begins with a proof of partial correctness (safety property). Without knowing safety properties, the proofs of liveness properties usually require more operational reasoning, which frequently leads to errors.

4. CONCLUSION.

The example shows that our method is useful even for small algorithms : it is sometimes difficult to find their invariants (see [4]). The need of an incremental methodology is still more pronounced for medium size or large programs. A problem which occurs frequently during the design of parallel processes is to know if some statement may be split or not. The methodology presented here provides reliable answers to such questions. If, in some case, a non-elementary statement may be split into several statements, these must be serialized in the right order. Our methodology helps to determine which order is acceptable. Mistakes of this kind occur frequently in parallel algorithms. It was the case for first versions of the "On-the-fly garbage collector" and the algorithm of Ricart and Agrawala previously mentioned (see

[8], [13] and [14]). Moreover, the study of these algorithms and others has made us believe that, contrary to a common view, the invariant method can be well adapted to parallel programming. Specifically, the size of the invariants remains acceptable.

The synthesis of invariants for concurrent programs has already been investigated. Clarke obtains the best invariant of a program as the least fixpoint of an equation written from the text of this program (see [2]). Another method has been developed by Manna and Pnueli (see [10]). Both methods apply only to a restricted class of parallel programs, which is not the case of ours.

REFERENCES.

[1] ASHCROFT, E.A., MANNA, Z., "Formalization of Properties of Parallel Programs", Machine Intelligence, vol. 6, pp. 17-41, 1970.

[2] CLARKE, E.M., "Synthesis of resource invariants for concurrent programs", ACM Toplas, Vol. 2, pp. 338-358, 1980.

[3] DIJKSTRA, E.W., "A discipline of programming", Prentice Hall, New Jersey, 1976.

[4] DIJKSTRA, E.W., "An assertional proof of a program by G. L. Peterson", EWD 779, 1981.

[5] GRIBOMONT, E.P., "Programmation parallèle", Internal Report, University of Liège, 1982.

[6] GRIBOMONT, E.P., "Proving parallel programs in an incremental way", submitted to Science of Computer Programming, 1983.

[7] GRIBOMONT, E.P., "Mutual exclusion in a computer network", submitted to Computer Networks, 1983.

[8] GRIES, D., "An Exercise in Proving Parallel Programs Correct", CACM, vol. 20, pp. 921-930, 1977.

[9] LEVIN, G.M., GRIES, D., "A Proof Technique for Communicating Sequential Processes", Acta Informatica, vol. 15, pp. 281-302, 1981.

[10] MANNA, Z., PNUELI, A., "Verification of concurrent programs : temporal proofs principles", Lecture Notes in Comp. Sc., vol. 131, pp. 200-252, Springer, 1981.

[11] OWICKI, S., GRIES, D., "An Axiomatic Proof Technique for Parallel Programs", Acta Informatica, vol. 6, pp. 319-340, 1976.

[12] PETERSON, G.L., "Myths about the mutual exclusion problem", Information Processing Letters, vol. 12, pp. 115-116, 1981.

[13] RICART, G., AGRAWALA, A.K., "An optimal algorithm for mutual exclusion", CACM, vol. 24, pp. 9-17, 1981.

[14] Corrigendum, CACM, vol. 24, p. 578, 1981.