WIS-CS-75-244

COMPUTER SCIENCES DEPARTMENT
University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706

Received March 25, 1975

ON PROGRAMMING DEPENDENCES BETWEEN PARALLEL PROCESSES

by

Gerald Belpaire

Technical Report #244

March 1975

#244

ABSTRACT

In order to study the problems of synchronization between concurrent processes, a distinction is made between the level of the formal models, the level of programming tools, and the level of the implementation practicalities. The model is seen as the semantics of the programming tools, and the implementation as the realization of the semantics at a lower level of abstraction.

Different classes of problems of synchronization are distinguished (e.g. exclusions, cooperations) and formal rules are defined to characterize them (the rules of dependence). The different classes are defined upon a common root of basic concepts and hypotheses, in order to be able to draw comparisons between them.

An important part of the work is dedicated to a formal treatment of the deadlock problem for the class of problems of exclusions between critical sections.

An attempt is made to give an answer to the question of understanding what characterizes a problem of synchronization, how it is related to the choice of programming primitives, and what are the effects of the implementation design.

This work was submitted in fulfillment of the requirements for the degree of "Docteur en Sciences Appliquées" at the Université Catholique de Louvain, Belgium.

Finally, the moral of this set of morals is : If you want to live happily, learn the tricks of the trade , put out plenty of solid uncontroversial (if possible dull) stuff, and avoid the qualms and pangs that accompany the search for depth - because this search is unending, for physics is bottomless : its foundations are temporary.

Mario Bunge, Foundations of physics.

ACKNOWLEDGEMENTS.

Unfortunate academic contingencies make of this thesis the work of one man. Reality is much different however and several names should be attached to this research.

Outstanding among them are Jean Pierre Wilmotte and Pierre L. Wodon for three years of collaboration at the University of Louvain. Undeniably, this research has progressed due to their ideas and efforts.

Thanks are also due to all those with whom we have discussed, either directly or indirectly, the subject of synchronization, to those who expressed, sometimes spontaneously, their interest in our work, and to those who were virulent critiques - for contradictions only can "transform ignorance into knowledge and limited knowledge into advanced knowledge"(Mao-Tsetoung) : Elie Milgrom (U.C.L.), Axel van Lamsweerde and Pierre J. Courtois (MBLE Research Lab.), G. Seegmüller (Universität München), Richard C. Lipton (Yale University), David L. Parnas (Technische Hochschule Darmstadt), Nico Habermann (Carnegie-Mellon University), Claude Kaiser (I.R.I.A. and Conservatoire National des Arts et Métiers, Paris), Michael Spier (Digital Equipment Corporation, Maynard, Massachusetts), Raymond Devillers (Université Libre de Bruxelles), Per Brinch Hansen (California Institute of Technology).

Finally, last but not least, financial assistance was provided by a common, but unvoluntary, effort of both Belgians and Americans taxpayers, during one year in Louvain and two years in Louvain-La-Neuve under a grant of the Belgian Government (S.P.P.S. Projet National d'Impulsion à la Recherche en Informatique, Contrat I-14.1/5), and since Summer 74 at the Computer Sciences Department and the Mathematics Research Center of the University of Wisconsin at Madison (for the latter under U.S. Army Contract DA-31-124-ARO-D-462).

For all these reasons, we will use the pronoun "we" to refer to the authors of these pages, not as a sacrifice to some stylistic convention but as a deliberate decision to reflect the reality.

CONTENTS.

1. PRELIMINARIES.

1.1 Introduction.

There has been in the past ten years a considerable amount of research carried out on the principles of synchronization: how they can be formalized, how synchronization problems coul be programmed, and how they should be implemented. The present work goes in the lines of these researches.

It is divided into four major parts. The first of them {2}† examines what are the basic concepts involved in synchronization problems and investigates the possibilities of defining a formal model of synchronization between parallel processes.

The second part {3} presents such a model for application to the class of problems of "exclusions" between accesses to critical sections, and to the problems of "cooperation" between processes. Both terms, "exclusions" and "cooperation", refer to well known problems described in the literature and they will be introduced in more details further in the text.

The third part {4,5,6} discusses the proper programming aspects of the problems of synchronization. Definitions of primitives (i.e. basic programming tools) and their implementation are given together with examples of their applications.

These developments are meant to explore the possibilities of integrating the three aspects of a comprehensive view of any computing problem : the theoretical model, the set of programming tools, and the implementation of these tools. Our aim is to use a coherent theoretical framework to define with precision the programming tools and to ensure the correctness of both their implementation and their utilization. As such , for the class of problems investigated here, we give an answer to the question of understanding what caracterizes a problem of synchronization, how it is related to the choice of primitives in some programming language, and what are the effects of the implementation design.

The last part of our work deals with deadlock problems. The classical way of handling deadlocks is to view them as resource allocation errors and therefore to study strategies of allocation that prevent these errors (e.g. [10,17,23,27]). The standpoint that we adopt here is to consider deadlocks as intrinsic properties of synchronization effects : i.e. a problem of synchronization will be deadlock-free, or not, because of its own characteristics independently of any resources allocation strategy. There are obviously possible relations between both approaches. This thesis brings a contribution to the second approach in the form of a necessary and sufficient condition of deadlock for general critical sections exclusions. This problem encompasses a wide class of synchronization problems (although not as large as the class studied by Devillers[17] as allocation problems).{7}.

Finally, conclusions and proposals for future research are presented.

1.2. Related work.

Most of the work on synchronization and communication between parallel processes can be divided into four categories :

1. The study of mathematical representations (models) of parallel computations.
2. The research in programming tools and techniques.
3. Experiments with the implementation of Operating Systems.
4. Deadlock problems.

---

†Throughout this text, figures between braces refer to particular sections of the thesis, and numbers between brackets refer to the bibliography.

1.2.1. Models of parallel computations.

Most of the work in this area is dedicated to investigations about the output functionality of parallel programs (i.e. how the final results of some parallel computation depends on the initial conditions). This led to a variety of interpreted and uninterpreted (program schemata) models.

These models are basically defined by a set of memory elements (having a value) and a set of computing elements (transformations upon these values). In the interpreted models, a meaning is assigned to the values and to the transformations (e.g. addition of integers). In the uninterpreted models no such meaning is associated to these elements ; only an algebraic relation is assumed to hold between them. Within such models, it is possible to focus the attention on the parallel control structures and to derive properties which hold under any interpretation. Properties of functionality, determinacy, and equivalence are examined under various constraints (hypothesis) on the model. Examples of such studies can be found in [30,31,42,47] .

The results achieved by these researches are not of direct application in this thesis, however the study of such theoretical questions proved to be very useful to gain more insight in the properties of parallel processes.

1.2.2. Programming Techniques.

The research in programming languages for parallel processes was founded several years ago by the introduction of Dijkstre's semaphores primitives [18,19,20] . Such approaches are motivated by a strict software engineering point of view : which tools does a programmer need to indicate the sections of a program to be executed in parallel, and to describe, with appropriate primitives, the needed synchronization among processes ?

Basically, the questions that have been investigated are :

1. A given problem being defined (most of them are unfortunately too simplistic), what are the programming primitives needed to draw an elegant solution to this problem, and is there any solution at all ? [5,13,20,22,49,51]

2. Which are the programming tools that can enhance if not the correctness at least an informal verification of parallel programs ? And particularly what kind of errors could be hopefully detected ? [5,6,12,1,2,20,22,52]

Most of the problems considered in these papers were in the fiels of "communication" between processes (e.g. the buffer problem) and of data shared ("shared data") by processes (e.g. mutual exclusions,Readers-Writers).

Because we have taken these programming problems for application of some theoretical principles,these works are closely related to our purpose. However it would be erroneous to think that we only want to propose "still another tool" among the many existing ones. Our claim is that answers to the practical questions stated hereabove can have several consequences that go beyond the simple definition of a new primitive. Indeed the imperatives of both conciseness and correctness verification can be better satisfied by the use of a new primitive rather than by intricate manipulations of existing ones. For this reason an important part of our thesis will be dedicated to these questions.

1.2.3. Operating Systems.

This third area of research is an example of practical applications of the implications of parallel processes research.

The notion of process is now universally used as a tool in the implementation of operating systems. Starting with Dijkstra's T.H.E. system, every modern operating system is presented as an hierarchical organization of parallel processes sharing resources and/or competing for them concurrently. Synchronization and communication problems are innumerable in the design of operating systems. Furthermore their complexity requires a stepwise approach to their understanding and to their (hopefully correct) design. This led to the development of the concept of level of abstraction and of hierarchical structures (cf. e.g.[37]).

Unfortunately, the many details of implementation proper to a particular system obscure the abstract properties of processes. It is therefore difficult to derive general principles from a close examination of Operating Systems. However two notable attempts to keep the design of a system to its essentials without being sunk into implementation details are due to Parnas and Price [34,35,36,40], and to Brinch Hansen [3,4].

Parnas's method is to use a system of definition for software modules. The specifications of the modules are done by means of the functions they realize and are independent of any implementation. Criteria to divide the system into modules are based on overall design decisions. Modules can be parametrized in order to fit a variety of computers and a variety of abstract properties (hence the name of "family" of operating systems).

Brinch Hansen's method is to design a programming language providing for the needed features for system design : concurrent processes, monitors, classes, synchronization tools. A systematic effort is done in order to facilitate compile-time checking of the communication between processes and to enhance a hierarchical structure :
"A hierarchical operating system written in concurrent Pascal will be tested component by component, bottom up (but could, of course, have been conceived top down or by iteration). When an incomplete operating system has been shown to work correctly (by proof or testing), a compiler can ensure that this part of the system will continue to work correctly when new untested program components are added on top of it. In a hierarchical operating system consisting of monitors and processes, programming errors within new components cannot cause old components to fail because old components do not call new components, and new components only call old components through well-defined procedures that have already been tested." [4].

1.2.4. Deadlocks.

The discussion of previous works on this matter is reported in section 7.

1.2.5.

The present work is related to these four areas of research although it never attempts to solve a decisive question of any of them. Its originality resides in an investigation of the relations between the four domains rather than of the properties valid within one of these domains.

It is an integrated approach wherein the theoretical principles are put in relation with practically relevant concepts and where the proposed programming features have a straightforward interpretation in the theoretical model. The ambition of such a large project is obviously limited by the scope of a thesis work, and also by our choice to study only the problems of synchronization.

2. PROCESSES AND EVENTS.

2.1. An example.

To introduce our purpose, we choose in the literature a typical example of what was at a time a basic issue of concurrent programming : to find solutions to problems stated in an informal manner (by "solutions" we mean the usual acceptation in the programming jargon : the solution of a problem consists of the description of this problem by means of some given programming tools.

In the present case the statement is taken from [13] :

"The processes of the first class, named writers, must have exclusive access as in the original problem [the one defined by Dijkstra], but the processes of the second class, the readers, may share the resource with an unlimited number of other readers."

"We demand of our solution that no reader be kept waiting unless a writer has already obtained permission to use the resource ; i.e.no reader should wait simply because a writer is waiting for other readers to finish."

The solution should use the primitives introduced by Dijkstra, whose definitions are taken from [18]:

"Definition. The P-operation is an operation with one argument, which must be the identification of a semaphore. (If "S1" and "S2" denote semaphores we can write "P(S1)" and "P(S2)".) Its function is to decrease the value of its argument by 1 as soon as the resulting value would be non-negative. The completion of the P-operation - i.e. the decision that this is the appropriate moment to effectuate the decrease and the subsequent decrease itself - is to be regarded as an indivisible operation."

"Definition. The V-operation is an operation with one argument, which must be the identification of a semaphore. (If "S1" and "S2" denote semaphores we can write "V(S1)" and "V(S2)".) Its function is to increase its value by 1 ; this increase is to be regarded as an indivisible operation."

(Dijkstra himself has given a different definition of the P-V-operation [19] but it does not matter which definition is used in this example.)

The solution given in [13] is :

```
integer readcount ; (initial value = 0)

semaphore mutex, w ; (initial value for both = 1)
```

READER	WRITER
P(mutex) ;	
readcount := readcount + 1 ;	
if readcount = 1 then P(w) ;	
V(mutex) ;	
....	P(w) ;
reading is performed	....
....	writing is performed
P(mutex) ;	....
readcount := readcount - 1 ;	V(w) ;
if readcount = 0 then V(w) ;	
V(mutex) ;	

This solution has its own merit as it was found at a time (1970) when it was believed that no solutions in terms of P and V was possible for this problem.

It is however clear that it presents several drawbacks :
1. Its complexity is disproportionate to the problem statement.
2. For this reason its understandability is limited and its correctness is far from obvious.

Why do so simple problems draw so complex solutions ? When a P-operation appears at the head of a critical section in a class A of processes, it prevents all classes of processes from entering the sections at the head of which the same P on the same semaphore occurs, including class A itself. Actually, one encounters these troubles every time that the synchronizing effect (i.e. closing the door) should be separated from the conditional waiting effect (i.e. passing the door).

To avoid such inconveniences, when using the P-V primitives, one needs to program intricate manipulations of variables of synchronization within mutually exclusive critical sections(the sections between P(mutex) and V(mutex) in this case). Considering the processes as belonging to some level of abstraction, the manipulation of the variables of synchronization belongs to a lower level of abstraction and therefore, should not appear explicitly in this program level.

This led us to start from a more elementary level, independently from the primitive operations, and to define the concepts of events and of relations of dependences between events. Then we propose new synchronization primitives that describe the needed - and only the needed - effect of some event upon some others. (Some of these results were published in [1,2]).

The basic questions studied in this thesis are related to the previous discussion. They are :

1. How to devise a theoretical framework within which relations between the problem statement and the solution can be established ?
2. Is it possible to classify the different problems of synchronization ?
3. What are the other primitives than P and V that one can consider and how to implement them ?

To give a flavour of what will follows this introductory example, we can introduce a very simple "formal" model of the preceding example.

Let a_1 (ask to write), a_2 (perform the writing), and a_3 (terminate the writing) be three actions performed in sequence by the writers. Let b_1, b_2, and b_3 be the corresponding actions for the readers.

Let $n(a_i,t)$ be the number of processes having performed the action a_i prior to time t .

It is easy to define the three synchronizing rules(that we shall in the future call <u>dependence</u> rules), that are underlying the informal statement given hereabove :

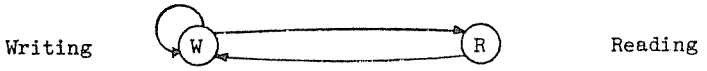
$$n(a_1,t) - n(a_3,t) > 0 \quad \rightarrow \quad a_1 \text{ and } b_1 \text{ cannot be executed;}$$

$$n(b_1,t) - n(b_3,t) > 0 \quad \rightarrow \quad a_1 \text{ cannot be executed.}$$

The three rules are of the same type :

> If some processes are executing one section (called critical section) of the program then no processes can start to execute another given critical section.

Such a "standard" rule will be called an exclusion and can be represented on a graph. For the three above exclusions this gives (see section {3} for more details) :



Now if we define some primitives to represent an arbitrary exclusion, it will be possible to program <u>any</u> problem of this type in a straightforward manner. (cf. {4} and {5} .).

The advantages of the method is essentially that the statement of the problem is given by the graph of exclusions, which is very simple (comparable to the simplicity of the problem) and then an expression in terms of some primitives is devised. The latter step can be almost automatic and, as a matter of fact, the kind of primitives used is not really important anymore provided that such correspondances between the primitives and the statement of the problem can be given.

2.2. Processes and events.

2.2.1. Motivation.

To proceed in our work, we need some basic formal definitions of the notions of process and of event. The aim of this section is to provide a justification of the way we have neglected parts of these widely used notions in order to focus our attention on the essential elements of these concepts that will be needed for the rest of this thesis.

It is important to notice that the model defined here is not intended either to give a perfect or complete image of the concrete problems of parallel process structures, or to give a large theoretical context within which decisive properties can be proven.

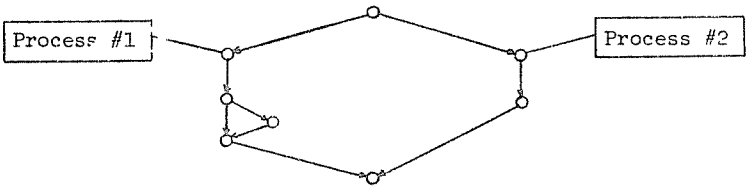
The ideas governing our choice are :

1. If one considers the concepts of process and of process structure (organization) currently in usage, what are the components one needs to retain into a model intended to represent only the synchronization among processes without oversimplifying the general concepts.

2. Which are the mathematical structures one should adopt to deal with problems of synchronization and how are these structures related to other mathematical models ?

2.2.2. Static and dynamic coordination of events.

A system of parallel processes will usually be defined as a partial ordering on a set of actions. Each individual process can be designated by a labelling (naming) of parts of the partial oreering structure :



It has been conjectured by Patil [39] (and illustrated on a particular example) that a partial ordering among actions cannot model all synchronizations (coordinations in his terms)† among processes. This conjecture is presented as the principal reason to use Coordination Nets (e.g. Petri Nets [26,39]). In these models actions are seen as events that can occur following certain rules (see {2.4.3.}). The conjecture is based on the idea that a partial ordering can only represent static (i.e. the rule does not depend on occurremces, although it specifies occurrences) coordinations among events. In some problems however, the coordination rule can change with the occurrences of particular events and therefore other structures than partial orderings are needed.

Continuing with conjectures, it seems reasonable to state that the Coordination Nets cannot models all types of synchronization. More evidence in favour of this statement will be given further. The model that we propose in this work is not a Coordinatuon Net model ; it is based on more general assumptions about the dynamic relations among events. For our present purpose, we will limit ourselves to a particular case of the general formulation ; the Petri Nets are an example of another particular case. It is dubious that one case can be reduced to the other({2.4.}).

It is evident that in practice the partial ordering among actions and events constitutes a primary notion of the static process organization (it corresponds to the "one instruction after the other" notion). For this reason, it appears explicitly in our model. An argument in favour of this decision is that Coordination Nets (that do not have an explicit partial ordering) are most suitable to represent hardware parallelism (in what is called asynchronous networks) but they are rather cumbersome for the representation of software parallelism (parallel processes and programs). The fact of keeping the static coordination as an explicit partial ordering diminishes the importance of this difficulty.

---

† In the rest of thsi thesis, coordination and synchronization will be used with the same meaning.

2.2.3. Processes and Programs.

The models of parallel processes are meant to represent computing activities as they occur dynamically. However, the programmer's view of the system is somewhat restricted to the program text itself and the comprehension of the system must be induced from the examination of the programs. A program is a mere description of what the processes and the coordinations among processes will be.

Moreover, from a theoretical point of view, it is of interest to be able to deduce properties of the system from the structures of the programs. These properties should be valid for any execution and therefore could be verified once forever. This is particularly interesting if several processes (possibly a large number) are executing the same piece of code. It is easier to prove some properites (e.g. the absence of deadlocks)by examining the structure of the program rather than by writing a model of each process executing the program.

The relation between a process and a program describing it is well understood and was outlined for the first time in a paper by Dennis and Van Horn [16] :

"A process is a locus of control within an instruction sequence. That is a process is that abstract entity which moves through the instructions of a procedure as the procedure is executed by a processor."

The standpoint adopted here is that programs should describe explicitly and in a simple way the coordination (synchronization) as represented in the theoretical model. At this condition only an easy comprehension of the problem can arise from a mere lecture of the program, and a straightforward relation can be established between programs and processes models. In our model, the partial ordering between actions is in a one-to-one correspondance with the sequence of instructions of a program, and to each synchronization - coordination action corresponds a primitive of the programming language.

2.2.4.Processes and Data Structures.

To obtain a complete representation of the process structures, one should make a distinction between the active part of a process (the operations seen as effectively performed) and the passive part of the same process (the memory elements whose values are transformed by this process).

In practice, the distinction appears already in the classical parameters passing for the call of a procedure in an algorithmic programming language ; the parameters being the passive elements and the procedural body being the active element. This mechanism can be extended to define dynamic binding of processes with data structures by means of "extended calling mechanisms" within operating systems ([40,45]).

The theoretical aspects of such problems have been very little investigated, although almost every model of parallel processes includes a representation for the active part (sometimes called the control network or structure) and a representation for data organization (data flow network). The reason why these attempts have been very little productive is that only a poor model of data structures is usually proposed (most of the time a heap of memory cells), and moreover, the problem of binding and calling mechanism is ignored.

If one examines this problem in the perspective of synchronization, one could assume a priori that the synchronization effect can be altogether dependent on the occurrences of the actions and on the resulting values of the performed operations. (In the same way as the execution of an if-then-else statement depends on the tested condition.).

There are example of process synchronization where such an interaction can occur : the decision to perform again the reading of some values when a parity error occurs.

A general model should therefore include the representation of such mechanisms. It will imply the formal definition of data structures and of the operations that a process can perform upon them, i.e, the model would have to be interpreted (e.g. by defining arithmetic operations, assignations, types). This is a problem whose solution is not known even for the pure sequential processes. Ultimately, the task must be done if one wants to provide an accurate model. It however falls beyond the possibilities of this thesis.

2.2.5. Summary.

It is now appropriate to summarize the discussion of this section.

A model of processes, in order to be complete, should consist of ;

1. A static structure of the actions of the processes, that will specify sequencing relations among the actions (e.g. a partial ordering).

2. A dynamic structure of coordination between the actions (e.g. a Petri Net).

3. For each process an active part (the control structure) and a passive part (the data structure).

4. In general the sequencing structure and the coordination structure can depend on the values of the data structure.

5. A model of relation between the processes and the programs that describe the structure of the processes.

Our model will involve only the items #1, #2, and #5 of this summary.

2.3. Basic definitions.

2.3.1. Programs.

The only thing that should be present in the model is the sequential control of the program. We are not interested, for the reason explained earlier, in giving a model which takes into account all the particularities of a program (if statement, loops, recursion...). The assumption is that a program can always be represented at first approximation as the description of some operations to be performed in a sequential order. The model can be refined if needed, and there is no contradiction between this formulation and other theoretical models. Moreover the operations of the programs are uninterpreted and they do not refer to explicit variables or values.

Since we consider only sequential programs, from now on the term "program" will be used for "sequential program". A program is defined by a sequence of instances of some operations :

S : the set of operations.

$\pi \in S^*$: a program. S^* is the free monoid defined on S.

2.3.2. Processes.

We consider sequential processes only, and we call them processes. They are seen as executions of programs (one execution of a program is one process). They are defined as sequences of instances of actions :

A : a set of actions.

$p \in A^*$: a process.

A program describes a process and each operation describes a corresponding action. We therefore assume a one-to-one mapping between S and A.

$A \overset{\phi}{\leftrightarrow} S$ ϕ is the relation of execution.

It is frequent that several processes execute the same program (in this case they form a class of processes). The representation given here does not allow for a distinction between them. The only way to distinguish processes within a class of processes is to name them differently. (This corresponds to some practical reality although a more sophisticated model, including data structures,could solve this problem.).

2.3.3. Events.

It is clear that a representation of the evolution in time of the processes is needed. To introduce it we define two sets :

T : a set whose elements are called instants (T is the time).

T is a totally ordered set i.e. for any two instants t_1, t_2 either $t_1 \leq t_2$ or $t_1 \geq t_2$.

E : a set whose elements are called events.

An event can occur at some instant of time. We represent the occurrence of an event by a couple :

$(e, t) \in E \times T$

Note that in this formalism an event can have several occurrences.

It is possible at the price of some additional complexity to define events as occurring during a certain lapse of time. Then an occurrence will be a couple :

(e, τ) where τ is an interval of T.

This will not be needed in this thesis.

The beginning and the end of an action in a process could be defined as events. However this would introduce some internal structure for the actions, since, if we consider a beginning and an end, there must be something in the middle. This level of detail is not needed for our purpose and we define the actions themselves to be events :

All instances of actions in a process are events (elements of E)

Let us now consider a process :

$p \in A^*, \quad p = a_1 a_2 a_3 \cdots a_n$

An evolution in time of p is a sequence of occurrences of actions (actions are events) :

$(a_1, t_1) \quad (a_2, t_2) \quad \ldots \quad (a_n, t_n)$

Two clauses are assumed to hold for any evolution in time of a process and for any set of evolutions of processes :

Clause of sequentiality. * If p is a process and p_t an evolution in time of this process :

$p = a_1 \ldots a_n \qquad p_t = (a_1, t_1) \ldots (a_n, t_n)$

then we have : $t_1 < t_2 < \ldots\ldots < t_n$ *

Clause of non simultaneity. * Let Π be a set of processes and Π_t the set of evolutions in time of these processes. For any two occurrences (a, t) and (a', t') of Π_t we have : $t \neq t'$ *

It is not necessary to have a clause of non simultaneity for any pair of events. It is introduced here for the sake of simplicity, and, anyway, it does not influence the validity of the results that we obtain. (about this, see the problem of indivisibility in section 4.2.3.).

2.3.4. Example.

Let p_1, p_2, p_3 be three processes defined on $A = \{a_1, a_2, a_3, a_4, a_5\}$

$p_1 = a_1 a_1 a_2 a_3 \qquad p_2 = a_5 a_4 a_3 a_1 \qquad p_3 = a_3 a_2 a_3 a_5$

An admissible evolution of these processes on the time T = N (the positive integers) is :

$(a_1, 1)\ (a_1, 5)\ (a_2, 6)\ (a_3, 10)$

$(a_5, 0)\ (a_4, 2)\ (a_3, 7)\ (a_1, 20)$

$(a_3, 3)\ (a_2, 40)\ (a_3, 41)\ (a_5, 50)$

2.4. Possibility and dependence.

2.4.1. The event possibility.

Let E_t be the set of occurrences of events prior to time t :

$$(e, t') \in E_t \quad \text{if} \quad t' < t$$

To any event e of E we attach a magnitude called the possibility of this event. This magnitude varies in time and the possibility of an event e at time t can be either possible or not-possible. Because the possibility is a two-valued magnitude, we can represent it as a predicate $P_e(t)$ whose truth value determines whether an event is possible or not at time t :

$$\text{(the possibility of the event } e \text{ is possible at time } t\text{)} \leftrightarrow P_e(t)$$

We say that the possibility of an event becomes possible at time t if :

$\sim P_e(t^-)$ and $P_e(t)$, where t^- is the instant immediately preceding t (or the limit of such instants).

Furthermore, the following rule applies to the possibilities of events :

If $(e, t') \in E_t$ then $P_e(t')$ (i.e. an event can occur only if its possibility is possible).

Thus an event can occur only if it is possible, but it does not have to occur. This corresponds to the usual sentence that says that, whenever an event becomes possible, it occurs within a finite time. An example of this can be found in the fact that when a semaphore passes from the closed state to the open state (e.g. its value goes from 0 to 1 in a P(s) operation), then the processes waiting in front of the semaphore can execute the operation within a finite time.

An example of application of the concept of possibility :

Let p be a process and p_t its evolution in time :

$$p = a_1\ a_2\ a_3\ \dots\ a_n \qquad p_t = (a_1, t_1)\ (a_2, t_2)\ \dots\ (a_n, t_n)$$

We know that the clause of sequentiality requires that :

$$\text{for every } i \text{ between } 1 \text{ and } (n-1), \quad t_i < t_{i+1}$$

This rule, which does not use the concept of possibility, can be given an equivalent formulation in terms of possibility :

Let $P_i(t)$ be the possibility of a_i at time t :

$$P_i(t) \leftrightarrow ((a_{i-1}, t_{i-1}) \in E_t \ \wedge (a_i, t_i) \notin E_t)$$

i.e. a_i can occur only if 1. it has not yet occurred, and 2. a_{i-1} has occurred.

2.4.2. Rules of dependences.

In principles, the predicate $P_e(t)$ can depend on arbitrary values or variables (e.g. a random variable). Because we decided (for the reasons explained earlier {2.2.4.}) that the coordination among events must depend only on occurrences of events (not on the results of operations performed when the events occur), it is natural to define $P_e(t)$ as a predicate on the set E_t of occurrences of events :

$$P_e(t) : E_t \rightarrow \{\text{true, false}\}$$

Within the possibility of an event, we want to precisely distinguish the part that only specifies the sequencing rule (which is of no interest since it is merely another expression of the sequential structure of the processes) and the part that will specify the coordination among the events.

Let $\Sigma_e(t)$ be the sequencing rule and $\Delta_e(t)$ a predicate such that :

$$P_e(t) = \Sigma_e(t) \wedge \Delta_e(t)$$

$\Delta_e(t)$ will represent the coordination. It is called the rule of dependence.

If we consider a set of processes, it is possible to define the rules of dependence among the actions of the processes. (Actions are defined to be events).

Let Π be a set of processes :

$$p_1 = a_1^1 \; a_2^1 \; \ldots\ldots \; a_{n_1}^1$$
$$p_2 = a_1^2 \; a_2^2 \; \ldots\ldots \; a_{n_2}^2$$
$$\ldots\ldots\ldots$$
$$p_m = a_1^m \; a_2^m \; \ldots\ldots \; a_{n_m}^m$$

and let Π_t be the evolution in time of this set of processes :

$$p_1(t) = (a_1^1, t_1^1) \ldots\ldots (a_{n_1}^1, t_{n_1}^1)$$
$$\ldots\ldots\ldots$$
$$p_m(t) = (a_1^m, t_1^m) \ldots\ldots (a_{n_m}^m, t_{n_m}^m)$$

The possibility of an action a_j^i at time t is defined to be :

$$P_j^i(t) = \Sigma_j^i(t) \wedge \Delta_j^i(t)$$

where :

$$\Sigma_j^i(t) = (((a_{j-1}^i, t_{j-1}^i) \in E_t) \wedge ((a_j^i, t_j^i) \notin E_t))$$

$\Sigma_j^i(t)$ is the clause of sequentiality. $\Delta_j^i(t)$ is called the rule of dependence (or dependence rule) of the action a_j^i upon the other actions of this set of processes. It is an arbitrary predicate on E_t.

The rules of dependence are taken as the formal definition of the coordination (or synchronization) among actions of processes. From a theoretical point of view there is no reason to make a distinction between Σ_j^i and Δ_j^i. This is justified only by practical considerations. We want to be able to represent in a comprehensive way the synchronization problems and to separate them from the pure sequential structure of the processes.

2.4.3. The Petri Nets as particular rules of dependence.

Petri Nets are defined in [26]and [39]. To show how they are only a particular case of rule of dependence we give briefly the definitions hereafter.

A Petri Net is a graph G with two kinds of nodes :

C the set of conditions A the set of events

AC a set of edges going from events to conditions

CA a set of edges going from conditions to events

A state of the net is represented by a number of markers placed in some conditions. We therefore define a condition to be an integer :

$$c_i \geq 0 \qquad c_i \in C$$

Transitions from one state to another are called firing of events. Let a_i be an event and

$$c_{i_1}, c_{i_2}, \cdots, c_{i_n} \quad \text{and} \quad d_{i_1}, d_{i_2}, \cdots, d_{i_m}$$

the conditions such that $(c_{i_k}, a_i) \in CA$ and $(a_i, d_{i_k}) \in AC$.

The event a_i can be fired if for every k, $1 \leq k \leq n$, $c_{i_k} > 0$. When the firing occurs, each c_{i_k} is decreased by one and each d_{i_k} is increased by one. Only one event can occur at a time.

A trivial Petri Net is :

It represents the sequencing rule of the actions of a process.

In the general case, the possibility of an event in a Petri Net is defined by :

$$P_i(t) = (\text{for every } k, \quad c_{i_k}(t) > 0) \qquad c_{i_k}(t) \text{ is the value of } c_{i_k} \text{ at time } t.$$

This is not an acceptable dependence rule since it does not relate the possibility to occurrences of other events. We can do so in the following way. An event a_i is said to have a negative effect on a condition c_j if (c_j, a_i) is an edge of the net. It is said to have a positive effect if (a_i, c_j) is an edge. The conditions :

$$c_{i_1}, \cdots, c_{i_n} \quad \text{and} \quad d_{i_1}, \cdots, d_{i_m}$$

defined earlier are the conditions upon which a_i has a negative and (resp.) positive effect.

We certainly have, if $c_u(t)$ is the value of some condition at time t :

$$c_u(t) = \sum_v \left(\alpha^+(a_v, c_u) - \alpha^-(a_v, c_u)\right) n_v(t) + c_u^0$$

$n_v(t)$ is the number of occurrences of a_v prior to time t (This is defined on E_t).

$\alpha^+(a_v, c_u) = 1$ if a_u has a positive effect on c_u, it is zero otherwise.

$\alpha^-(a_v, c_u) = 1$ if a_u has a negative effect on c_u, it is zero otherwise.

c_u^0 is a constant giving the initial value of c_u .

If we put this last relation in the expression of the possibility, we obtain a relation between the possibility and the number of past occurrences of some events. This is clearly a rule of dependence. It is actually a particular case of the general rules of dependence (more exactly a class of particular cases). The Petri Nets are a simple notation to represent problems of this class.

It is easy to prove that for any condition at any time we have $c_i(t) \geq 0$. Indeed, to become negative a condition must first vanish in some state, and second, in that state an event having a negative effect on the condition must be fired. But this is a contradiction because, if a condition vanishes, no event having a negative effect upon it is possible.

Said otherwise, if a condition c_{i_k} determines the possibility of an event a_i then necessarily $\alpha^-(a_i, c_{i_k}) = 1$.

Another particularity of Petri Nets is that the sequencing rule is not separated from the dependence rule. If one wants to decompose a Petri Net in several sequential processes, one has to make a choice between several possibilities(there is always one (trivial) decomposition where each event is a one-event process).

2.4.4. The conjunction law.

We define now another example of dependence rule that will be used in section 3. The conjunction law is easily defined in the following terms :

Let A be a set of events : $a_1, a_2, \ldots, a_n$

For each pair a_i, a_j of events, we define a integer valued condition $c_{ij}(t)$:

$$c_{ij}(t) = n^+_{ij}(t) - n^-_{ij}(t) + n_{ij}$$

$n^+_{ij}(t) = \alpha^+_{ij}\ n_i(t)$ is the number of positive effects of a_i on a_j .

$\alpha^+_{ij} = 1$ if a_i has positive effects on a_j ; $n_i(t)$ is the number of occurrences of a_i before t.

$n^-_{ij}(t) = \alpha^-_{ij}\ n_i(t)$ is defined in the same way but for negative effects.

n_{ij} is a constant.

The possibility of a_j is defined :

$$P_j(t) = \Sigma_j(t) \wedge (\text{for each } i,\ 1 \leq i \leq n,\ c_{ij}(t) > 0\ )$$

One can verify that there is no Petri Net that can be equivalent to a general case of conjunction law. Indeed we do not have here a restriction on the events that can have negative effects on the conditions, and in particular, it is possible, for some conditions at some time t, that :

$$c_{ij}(t) < 0$$

2.4.5. The majority law.

The majority law is the most important rule of dependence defined in this work. It will be used thoroughly for the definition of the d-operation ({ 4 }) and for their implementation.

Using the notations of the preceding section ({2.4.4.}) we can write the possibility of an event a_j :

$$P_j(t) = \Sigma_j(t) \wedge \left(\sum_i n^+_{ij}(t) - \sum_i n^-_{ij}(t) + n_j \geq 0 \right)$$

n_j is a constant proper to a_j .

The general Petri Nets coordination is a combination of both the conjunction and the majority laws,with some restrictions on the terms of the summation.

The choice to use either one or the other law in practical problems cannot be made in an absolute theoretical manner. It will depends on practical considerations on the programming tools and on their implementation.

2.4.6. Possibility and necessity.

The possibility $P_e(t)$ of an event e at time t is used in a statement of the form :

"If $P_e(t)$ is <u>possible</u> then the event e <u>can</u> occur at time t

or within a finite time after t."

This statement involves a logical modality bearing on the occurrence of e. In the present case this modality is the possibility. It is conceivable however to define other modalities, in particular the <u>necessity</u> :

"If $P_e(t)$ becomes <u>necessary</u> at time t then the event e <u>must</u>

occur at the instant t^+ (the instant following t)."

In this case $P_e(t)$ is the necessity and it can have the value <u>necessary</u> or <u>not-necessary</u>.

The latter modality involves completely different synchronization problems than the ones studied in this thesis. As a remark, we can say that synchronizing tools such as interrupts involve the modality of necessity. This is why usual synchronizing primitives are not suitable to program real-time systems where responses to some occurrences of events must be necessary and not simply possible.

It is important to note that the necessity for an event to occur does not override the clause of non-simultaneity. In any case two events cannot occur at the same time. If one event must occur, its occurrence will be delayed if needed till a "free" instant is available for its occurrence. This situation arises in practice whenever several interrupts occur at the same time but only one of them is granted.

3. EXCLUSIONS AND COOPERATIONS.

3.1. Exclusions.

3.1.1. Critical sections.

In the literature critical sections are parts of programs, that are subject to some kind of special rules of execution. In our formalism, we can give the following definition of critical sections.

Let us consider a program on a set of operations S :

$$\pi \in S^* \qquad \pi = b_1 b_2 \ldots b_n \qquad \text{the } b_i \text{ are instances of operations of } S$$

If several processes are executing the program π the number of times that any particular instance of an operation would have been executed prior to an instant t is given by the number of occurrences of the action corresponding to that particular instance prior to time t . If we call $a_i = (b_i)$ the action corresponding to b_i , the number of executions of b_i prior to time t will be $n_i(t)$ as defined in section 2.4.3.. †

We define now $[a_i]_t$ as the number of processes having executed b_i but not yet b_{i+1} at time t :

$$[a_i]_t = n_i(t) - n_{i+1}(t)$$

We define now an interval of the program π (or of the processes executing π) as :

$$[a_{i_1}, a_{i_2}] = a_{i_1} a_{i_1+1} \cdots a_{i_2} \qquad \text{with } i_1 \leq i_2$$

The population of an interval is :

$$[a_{i_1}, a_{i_2}]_t = \sum_{k=i_1}^{i_2} [a_k]_t \qquad [a_{i_1}, a_{i_1}]_t = [a_{i_1}]_t$$

A critical section of a program is an interval of that program.

3.1.2. Relations of exclusions.

We consider now a set of programs :

$$\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\} \qquad \pi_i = a_1^i a_2^i \ldots a_{n_i}^i$$

and a set of critical sections of this programs :

$$I = \{I_1, I_2, \ldots, I_m\} \qquad I_k = [a_{j_k}^{i_k}, a_{\ell_k}^{i_k}] = I_{j_k \ell_k}^{i_k}$$

$a_{j_k}^{i_k}$ is the initiation of the critical section I_k, i.e. upon execution of the operation, a process will be "in" the critical section.

A relation of exclusions on I is a binary relation :

$$R \subseteq I \times I \qquad R_{ij} = (I_i, I_j), \quad R_{ij} \in R$$

We will use a graph notation to represent the relations of exclusions.

---

† In the rest of this work, actions will be the only events that we consider. We therefore use without confusion the expression "possibility of actions" and "occurrences of actions". Furthermore the actions are in one to one correspondance with the operations and we use the same notation for both actions and operations.

Intuitively, an exclusion (i.e. an element of the relation of exclusions $R_{uv} = (I_u, I_v)$) means that no process is allowed to enter the critical section I_v whenever there are some processes in the section I_u . To define this rule formally we associate to each exclusion a rule of dependence :

$$[I_u]_t > 0 \leftrightarrow \text{(the initiation of } I_v \text{ is excluded at time } t\text{)} \leftrightarrow \sim\Delta^{i_v}_{j_v}(t) \quad \text{, if} \quad I_v = [a^{i_v}_{j_v}, a^{i_v}_{\ell_v}]$$

This is an equivalence between three predicates. The second one is an informal sentence that we use to designate the formal rule. The first and the third ones can be written :

$$\sim \Delta^{i_v}_{j_v}(t) = [I_u]_t > 0$$

$\Delta^{i_v}_{j_v}(t)$ is the rule of dependence determining the possibility of $a^{i_v}_{j_v}$. Said otherwise, this rule (and therefore the exclusion R_{uv}) states that the possibility of $a^{i_v}_{j_v}$ is <u>not-possible</u> when the population of I_u is positive.

This rule of dependence is defined on the set E_t of the occurrences of past events because the population of any interval is defined on the set E_t .

3.1.3. Exclusions and the majority law.

It is easy to show that the rule of dependence defining the exclusions is a particular case of the majority law.

Let $(I_u, I_v) \in R$. As done in section 2.4., we can assign to the action $a^{i_u}_{j_u}$ a negative effect on the possibility of $a^{i_v}_{j_v}$, and to $a^{i_u}_{\ell_{u+1}}$ a positive effect on the same possibility. Then :

$$[I_u]_t = n_\mu(t) - n_\nu(t) = n^-_{\mu\rho}(t) - n^+_{\nu\rho}(t)$$

if we define $\mu = (^{i_u}_{j_u})$, $\nu = (^{i_u}_{\ell_{u+1}})$, $\rho = (^{i_v}_{j_v})$ to simplify the notations.

Since the exclusion on I_v is determined by all I_u such that $R_{uv} \in R$, we have

$$\text{(there si a } u \text{ , } [I_u]_t > 0 \text{)} \leftrightarrow \sim\Delta^{i_v}_{j_v}(t)$$

or equivalently, because $[I_u]_t \geq 0$ for all u and t :

$$\sum_u [I_u]_t > 0 \leftrightarrow \sim\Delta^{i_v}_{j_v}(t) \text{ , the sum is made on all } u \text{ such that } R_{uv} \in R$$

From this and from the above relations we obtain easily :

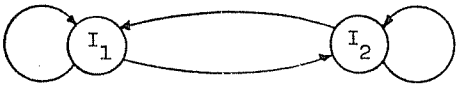
$$\sum_u [I_u]_t \leq 0 \leftrightarrow \sum_u n^+_{\nu\rho}(t) - \sum_u n^-_{\mu\rho}(t) \geq 0$$

This is a majority law on the possibility of $a^{i_v}_{j_v}$. Because the population of an interval is always greater or equal to zero, it is possible to define the exclusions in terms of the conjunction law. The interest of defining them by a majority law is that there is an efficient implementation for the operations that will be used to program the majority law. This is an example where practical considerations determine the choice between two equivalent theoretical representations.

3.1.4. Example.

$$\pi_1 = a_1^1\, a_2^1\, a_3^1 \qquad \pi_2 = a_1^2\, a_2^2\, a_3^2$$
$$I_1 = I_{12}^1 \qquad I_2 = I_{12}^2$$

The relation of exclusions is :



This a formal expression of the mutual exclusion betwwen I_1 and I_2. Indeed it is easy to prove that only one process can be executing I_1 or I_2 at any time. If $[I_1] > 0$ then no process can enter I_2 i.e. $[I_2]$ cannot increase. The same is true for $[I_1]$. Initially, $[I_1] = [I_2] = 0$. As soon as one population , say $[I_1]$, becomes equal to one, neither $[I_1]$ nor $[I_2]$ can increase. Thus $[I_2]$ stays equal to zero and $[I_1]$ cannot become equal to two.

3.2. Cooperations.

3.2.1. Relations of cooperations.

Another practical example of special rule of execution between critical sections is the relation of cooperations. It is used in practice to solve problems where processes are producing and consuming (consumable) resources. A classical example is the finite buffer problem ({5.6.}). In this class of problems, the coordination does not depend on the number of current executions of some critical sections, but on the total number of past and present executions of the critical sections. It is therefore conceptually different from the exclusions. Formally we have :

Let I_u, I_v be two critical sections. We know that :

$$[I_u]_t = n_\mu(t) - n_\nu(t) \text{ , if } I_u = I_{j_u \ell_u}^{i_u},\ \mu = (_{j_u}^{i_u}) \text{ , and } \nu = (_{\ell_u+1}^{i_u})$$

and a similar relation holds for I_v.
A cooperation of I_v with I_u is a condition on the number of past executions of $a_{j_v}^{i_v}$ (the initiation of I_v), and of $a_{\ell_u+1}^{i_u}$ (the termination of I_u). This condition must hold at all time :

$$\text{for all } t \text{ , } n_\rho(t) \leq n_\nu(t) + n_{uv} \text{ , where } \rho = (_{j_v}^{i_v}) \qquad (321\text{-}1)$$

n_{uv} is an integer constant. Intuitively, this relation can be interpreted as :

> The number of terminations (completed executions) of I_v cannot exceed the number of initiations (current or completed executions) of I_u augmented by a constant.

Such cooperations can be defined on a set of critical sections. We obtain a relation of cooperations. It can also be represented on a graph (cf. {5.6.}).

3.2.2. Cooperations and the conjunction law.

The relation (321-1) defines a limit on the number of executions of $a_{j_v}^{i_v}$. This can be expressed by a rule of dependence on the possibility of $a_{j_v}^{i_v}$. If we assign to $a_{\ell_u+1}^{i_u}$ a positive effect on the possibility of $a_{j_v}^{i_v}$, and to $a_{j_v}^{i_v}$ a negative effect on its own possibility, the relation (321-1) is written :

$$\text{for all } t\ ,\ n^{+}_{\nu\rho}(t) - n^{-}_{\rho\rho}(t) + n_{uv} \geq 0$$

because : $n^{+}_{\nu\rho}(t) = n_{\nu}(t)$ and $n^{-}_{\rho\rho}(t) = n_{\rho}(t)$. The possibility of $a^{i_v}_{j_v}$ is now defined :

$$P_{\rho}(t) = \Sigma_{\rho}(t) \wedge \Delta_{uv}(t) \qquad \text{since } \rho = (^{i_v}_{j_v})$$

$$\Delta_{uv}(t) \quad (n^{+}_{\nu\rho}(t) - n^{-}_{\rho\rho}(t) + n_{uv} \geq 0)$$

If a relation of cooperations R is defined on a set of critical sections, the possibility of $a^{i_v}_{j_v}$ is defined :

$$P_{\rho}(t) = \Sigma_{\rho}(t) \wedge (\text{for all u such that } R_{uv} \in R,\ \Delta_{uv}(t))$$

This is a particular case of conjunction law. The relation of cooperations can therefore be expressed as a conjunction law on the possibility of the initiations of the critical sections.

3.2.3. Interpretation of the cooperation.

If we replace "completed executions of I_v" by "portions consumed by process p_v" and "current or completed executions of I_u" by "portions being currently produced or already produced by process p_u", we have a new statement of the copperation :

> The number of portions consumed by p_v cannot exceed the number of portions being currently produced or already produced by process p_u, augmented by a constant.

This is a statement of the cooperation through an infinite buffer [18]. The constant is the number of initially available portions. Note that the statement allows process p_v to consume a portion not yet completely produced. This suggests that the relation of cooperation is not sufficient to guarantee a proper coordination between p_v and p_u. Indeed, in practice, the access to the buffer will be made mutually exclusive by some relation of exclusions between I_v and I_u. This example shows how the formalism can reveal details that are intuitively left implicit.

4. THE D-OPERATIONS.

4.1. Motivation.

4.1.1. Models vs. Programming Languages.

An abstract model of synchronization, even if it is not used as a deductive system, can be useful both as a definitional tool and as an abstract semantic interpretation. As a definitional tool it has the advantages of conciseness and of formal character ; as a semantic interpretation, it is a clearly delimited subset of the natural language in which our comprehension of the problem is defined. A model of some phenomenon can be seen as a language in which the semantics of the considered phenomenon is expressed.

Usually a model cannot be used as a programming language. Not only due to technological reasons (e.g. no compiler is available) but also due to more fundamental reasons. A programming language is used to build descriptions of what processes are to execute while the model is usually a representation of the execution itself. There are concepts represented in the model which are not present in the programming language e.g. states, transitions, events, occurrences... Moreover a programming language usually requires features that facilitate the task of the programmers and match some operational objectives : implementation, error detection... These features are absent of the model.

In the area of synchronization, a typical "model" without programming counterpart is the Petri-nets formalism[26,39]. And a programming language without corresponding "model" consists of the critical regions and monitors of Brinch Hansen and Hoare[3,8,25]. An effort to present both aspects comprehensively can be found in [43].

4.1.2. Defining primitives.

The theoretical model developed hereabove {2.3} can be considered as a model in which problem of synchronization can be defined and understood. It is however very impractical as a programming language and we need to define programming tools that will verify the characteristics of a programming language. This procedure is not unlike the problem of defining ALGOL 60 assuming that it must be suitable for numerical computations.

Usually, in the literature, synchronizing primitives are defined as some manipulations of variables (i.e. a pure operational definition, and sometimes only an implementation). Their semantics is only suggested by an example. The d-operations are defined syntactically (as a structured notation) and semantically (as corresponding to some rule of dependence).

A simple problem must have a simple representation in terms of the rules of dependences (otherwise the model would be inadequate) and, if the rules of dependences have a simple programming counterpart, the program corresponding to a simple problem should be simple as well. We will see in section 5. that this is indeed the case. Sometimes in the literature simple problems were giving birth to intricate solutions. Therefore this result should not be underestimated.

4.1.3. Further requirements.

Only recently (cf. e.g. [5]), the problem of designing synchronizing primitives enhancing the possibility of detecting programming errors was investigated seriously. This problems is analog to the detection of programming errors at compile-time by introducing some redundancy in the language. It is a well-known problem in "sequential" programming that has been little investigated in concurrent programming. This work is not oriented toward these researches.

4.2. Definitions.

4.2.1. Notations.

The operations that are defined here are extensions of the primitives defined by Dijkstra [18,19,20] . They are meant to describe rules of dependence and for this reason we call them d-operations. The general definition proceeds as follows :

Let $x_1, x_2, \ldots, x_n$

be a vector $\underline{x}$ of n integer variables that we call semaphores.†
A d-operation is defined by :

1. A predicate $p(\underline{x})$ on the variables $x_1, \ldots, x_n$

2. A function $f(\underline{x})$ on the same variables. The domain of f is therefore Z^n and its co-domain is defined to be Z^n as well. (Z is the set of integers).

The notation for the operation is : ††

$$p(\underline{x}) : \underline{x} \leftarrow f(\underline{x})$$

4.2.2. Execution of the operation.

The execution of the operation is an event and its possibility is defined by a sequencing rule and a dependence rule :

$$P(t) = \Sigma(t) \ \wedge \ \Delta(t) \qquad \text{with} \qquad \Delta(t) = (\ p(\underline{x}) \text{ is true at time } t\ )$$

Whenever the execution occurs, its effect is to compute $f(\underline{x})$ and to assign the result to $\underline{x}$.

An example of a d-operation is the P(s) of Dijkstra (do not confuse with P(t), the possibility) :

$$P(s) = (s > 0) : (s \leftarrow s-1)$$

Another (irrealistic) example might be :

$$((x_1+x_2 < x_3+x_4) \wedge (x_5 \text{ is odd})) : ((x_1,x_2,x_3,x_4,x_5) \leftarrow \underline{\text{if}}\ x_1 < x_3\ \underline{\text{then}}\ (x_5,x_4,x_3,x_2,x_1)\ \underline{\text{else}}\ (0,0,0,0,0)\ \underline{\text{fi}})$$

4.2.3. Indivisibility.

When executed an operation becomes an event whose occurrence is seen as instantaneous and non-simultaneous with any other event {3} . This theoretical assumption is sufficient to guarantee that no undefined behavior will arise from concurrent accesses to the variables $\underline{x}$. In practice however, this property must be verified by the implementation of the operation. This requirement is called the indivisibility of the operation (inseparability in [50]). Its operational function is to prevent all interferences between accesses to te same variables by different processes, at the level of the implementation.

---

† The name "semaphore" was originally used to designate both the variables and the operations performed upon them. We use it only for the variables.

†† The variables $\underline{x}$ have no meaning whatsoever in this definition. The predicate $p(\underline{x})$ will express the possibility of some event and therefore the variables $\underline{x}$ will be related to some variables defined on the set of occurrences of events (E_t in our notation). In simple cases they are actual values of the number of past occurrences of some events ("counters"). One may argue that they should not appear if one wants to respect the principles of definitions {4.1.2.} . It is indeed possible to do so in simple cases. The theoretical status of these so-called "control variables" is quite intriguing and they are a subject of interest for amateurs of philosophical discussions.

This notion of indivisibility was introduced for the definition of the P and V [18] and it is acknowledged to be sufficient for a correct implementation of the operations. It is not known to which extent it is necessary (i.e. whether parts of the execution of the operation can be interleaved or made truly concurrent). All the operations used in this work are assumed to be indivisible.

4.2.4. Particular d-operations.

The operations that are mostly used in this thesis, for which examples are given together with an implementation and for which the theorems of deadlocks are proven, are particular cases of d-operations, defined as follows :

Their general form is :

$$s_1:s_2:\ldots:s_n:\underline{\text{down}}\ t_1,\ldots,t_m\ \underline{\text{up}}\ u_1,\ldots,u_k \qquad \text{with } n \geq 0\ ,\ m \geq 0\ ,\ k \geq 0 \qquad (424\text{-}1)$$

Some of the variables $s_1,..,s_n,t_1,..,t_m,u_1,..u_k$ can be identical.

This is a notation for the operation :

$$(\ \textstyle\sum_i s_i \geq 0\ ):(\underline{t} \leftarrow \underline{t}-(1,1,..,1)\ ;\ \underline{u} \leftarrow \underline{u} + (1,1,..1)\ )\ ^{\dagger}$$

This definition includes the three following elementary d-operations (introduced by Wodon [51]) :

s: the passage, $\underline{\text{down}}$ t the closing, $\underline{\text{up}}$ u the opening

The general operation (424-1) can be formed by a rule of composition on the three elementary operations. This rule is purely syntactical, exactly as the syntactic rule producing a program.

The semantic of the operation (424-1) is defined in our theoretical model by a majority law. Let

$$a_1,a_2,\ldots,a_N$$

be N actions to be coordinated by a majority law determining their possibilities.

If a_j has a negative effect on the possibility of a_i , we define a_j as a d-operation involving a semaphore s_{ji} as closing operand :

$$\underline{\text{down}}\ s_{ji} \qquad (424\text{-}2)$$

and a_i involves s_{ji} as a passage operand :

$$s_{ji}:$$

In the same way, if a_k has a positive effect on a_i the d-operation of a_k will involve an opening operand :

$$\underline{\text{up}}\ s_{ki} \qquad (424\text{-}3)$$

And a_i a passage operand :

$$s_{ki}:$$

The operation corresponding to the action a_i will have the form :

$$s_{k_1i}:s_{k2i}:s_{k_3i}:\ldots:s_{k_ni}:\underline{\text{down}}\ \ldots\ \underline{\text{up}}\ \ldots$$

---

†Cases like (s,s) ← (s+1, s+1) are defined to be equivalent to s ← s+2 .

Its possibility is defined by :

$$\Delta_i(t) = \left(\sum_u s_{k_u i} \geq n_i \right)$$

where : $s_{k_u i} = n^+(a_i, a_{k_u}, t)$ or $s_{k_u i} = n^-(a_i, a_{k_u}, t)$

if (424-3) or (424-2) is applicable (resp.).

The constant n_i is proper to each operation a_i and determines the initial possibility. These values are attributed to a sum of terms. To deduce from them the initial values of each semaphore variable we need to solve a system of linear equations on the integers. In the simple cases, this system is trivial {5, Appendix}.

4.2.5. Adequacy and Reliability.{4.1.2., 4.1.3.}.

It is important to notice that the elementary operations together with their law of composition form a programming language with its primitives and its syntax. The semantics of this language is defined by an interpretation in terms of dependence rules between the executions of the operations (which are events). This language is so far very elementary but the possibility of defining more complex structures exists.(In particular meaningless or incorrect operations can be constructed.).

This fact has several important consequences in regard to the programming of dependences between parallel processes :
1. The problem of the adequacy of the language must be studied. The choice of the primitives (elementary operations), of the control structures (laws of composition), and of the data structures (semaphores) can be discussed in regard to their ability to provide elegant solutions to programming problems. This has been the subject of several proposals for new primitives ([1,5,49,52]).
2. The problem of the definition of both its syntax and its semantics is relevant. This is the object of this research (for other approaches, see e.g. [43]).
3. As said before, the problem of its reliability (errors detection) is debatable as well.
4. The synchronizing aspect of the language is separated from its algorithmic aspect. Both aspects can be seen as sublanguages with different syntaxes and semantics, that are merged to form a tool for programming parallel processes. This aspect of the problem appears very clearly in the research on Concurrent Pascal {3} .

4.2.6. Other operations.

As a consequence of the preceding discussion, it is possible to define an enormous variety of synchronizing primitives. As an example, we suggest here another law of composition for the three elementary operations defined hereabove. It is a representation for the conjunction law that will be used later on an example of programming problem.

The notation is :

$$s_1 * s_2 * \ldots * s_n : \underline{\text{down}}\ t_1, \ldots, t_m\ \underline{\text{up}}\ u_1, \ldots, u_k$$

Some of the variables can be identical. This is a notation for the operation :

$$p(\underline{s}) : (\underline{t} \leftarrow \underline{t} - (1,..,1)\ ;\ \underline{u} \leftarrow \underline{u} + (1,..,1)\ )\ \dagger$$

where :

$$p(\underline{s}) = \left((s \geq 0) \wedge (s_2 \geq 0) \wedge \ldots \wedge (s_n \geq 0) \right)$$

The semantics of the operation as interpreted in the model of dependences rules is a conjunction law.

---

† Cases like $(s, s) \leftarrow (s+1, s+1)$ are defined to be equivalent to $s \leftarrow s+2$

4.2.7. P(s) and V(s).

Most of the operations defined in the literature are particular cases of the general form defined here {4.2.1.} . For instance, we can define the P-V operations of Dijkstra ([18,20] but not [19]).

$$P(s) \equiv (s > 0) : (s \leftarrow s - 1)$$

$$V(s) \equiv (T) : (s \leftarrow s + 1)$$

(T) is the tautology (always true predicate).

4.2.8. P(n,s,) and V(n,s).

These operations are defined in [49] :

$$P(n,s) \equiv (s \geq n) : (s \leftarrow s - n)$$

$$V(n,s) \equiv (T) : (s \leftarrow s + n)$$

5. EXAMPLES.

5.1. The first problem of readers-writers [13] .

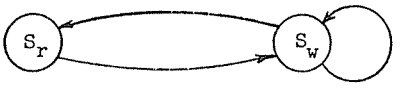
This problem was introduced earlier {2,3} . The definition proceeds as follows.

The exclusions are :

1. If a reader is reading then no writer can begin to write.
2. If a writer is writing,then no reader can begin to read.
3. If a writer is writing then no other writer can begin to write.

The critical sections, the relation of exclusions, and the graph of exclusions are respectively :

$S = \{S_r, S_w\}$ $\quad R = \{R_{rw}, R_{wr}, R_{ww}\}$



The program written with the d-operations is obtained by using a down on some semaphore to program a negative effect on some possibility, and a up to program a positive effect. A detailed procedure is given in [1]. It is a systematic application of the definitions given in sections 2.4.5., 3.1.3. and 4.2.4..

(readers) s_{wr}:down s_{rw} ;
reading ;
up s_{rw} ;

(writers) s_{rw}:s_{ww}:down s_{wr},s_{ww} ;
writing ;
up s_{wr},s_{ww}

Every semaphore is initialized to zero (because the possibility is initially possible).

This program can be simplified ; the semaphores s_{ww} and s_{wr} are equivalent (in some sense {Appendix}). Indeed their values are always equal because :
1. as opening and closing operand they always appear together ;
2. they are initialized to the same value.
The above program can thus be reduced to :

s_{ww}:down s_{rw} ;
reading ;
up s_{rw}

s_{ww}:s_{rw}:down s_{ww} ;
writing ;
up s_{ww}

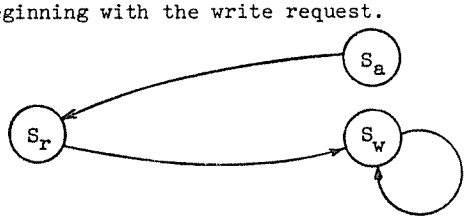
5.2. The second problem of readers-writers [13].

This problem is the same as the first one except that the third exclusion is replaced by :

3'. If a writer is asking to write then no reader can begin to read.

Thus we have a third critical section surrounding S_w and beginning with the write request.

$S = \{S_r, S_a, S_w\}$ $\quad R = \{R_{rw}, R_{ww}, R_{ar}\}$



The program is :

(Readers)

s_{ar}:down s_{rw} ;
reading ;
up s_{rw}

(writers)

down s_{ar} ;
s_{ww}:s_{rw}:down s_{ww} ;
writing ;
up s_{ww} ;
up s_{ar}

At this stage, we can apply a reduction rule that is the dual of the one used for the first problem {5.1}.This rule is developed in the Appendix. Its application gives the program :

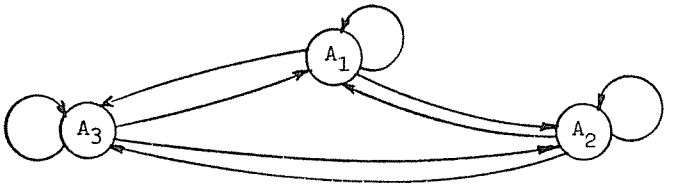
s_{ar}:down s_{ww} ;
reading ;
up s_{ww}

down s_{ar} ;
s_{ww}:down s_{ww} ;
writing ;
up s_{ww} ;
up s_{ar}

Remark : One can see in the two preceding examples that,by using the d-operations, we avoid the mutual exclusion sections usually found in solutions with P-V (e.g. the P(mutex)...V(mutex) in [13,20] and in {2.1.}). We find these mutex-sections troublesome for two reasons. First, they are irrelevant to the definition of the problems to solve, and therefore, they introduce some additional complexity to the solutions. Second, the synchronizing effect of the P(mutex)...V(mutex) belongs to another level of abstraction than the synchronization required by the definition of the problems.

5.3. Concurrent control with n priority classes [49].

There are n classes of processes with one critical section for each class : A_i , $1 \leq i \leq n$. There is an arbitrary relation of exclusions between the A_i, represented by some graph of exclusions. To concretize the problem, we take an example of mutual exclusion with three classes of processes :



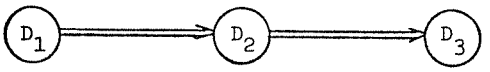
There are additional rules of execution of the critical sections, establishing priorities between the classes of processes :

> When a process of the class j is asking to execute A_j ,then no process of any class i, i>j can ask to execute A_i

Thus we must add n critical sections D_i with each D_i surrounding the corresponding A_i . The relation of exclusions is :

$$R = \{R_{ij} = (D_i, D_j) \mid 1 \leq i < j \leq n \}$$

This relation establishes a total ordering on the D_i and the graph of exclusions can be abbreviated by :



Now the program for the class # i can be written :

(Priority class # i)

s_i:down $s_{i+1}, s_{i+2}, \ldots, s_n$;

Critical-section-A_i ;

up $s_{i+1}, \ldots, s_n$

Remark : When we apply the preceding construct to the second problem of Readers-Writers, we obtain a solution different from the one given hereabove. However we could accept the solution as a "correct" solution depending on our comprehension of the intuitive statement {2.1.} of the problem.
This shows the possibility of confusion arising from the usage of high level intuitive statements and illustrates the need for precise definitions and representations of exclusions. The second problem of Readers-Writers can be generalized to give a different definition of a problem of priority classes.

The program obtained in this case is :

(readers)

s_2:
3
s_{ww}:down s_{rw} ;

reading ;

up s_{rw}

(writers)

down s_2 ;
1
s_{ww}:s_{rw}:down s_{ww} ;

writing ;
2
up s_{rw} ;

up s_2

It is the first problem of readers and writers plus a priority rule of the writers on the readers. There is a phenomenon (that we called "sneaking in") which can happen in this solution and not in the one given in section 5.2.. If the process *2* exits the "writing" critical section, the process *3* can enter the "reading" critical section before *1* can enter the"writing" critical section. The priority rule is not a priority of access to the critical sections but a priority of demand of access. From this comes the difference between the two solutions.

5.4. The problem of the five dining philosophers [20] .

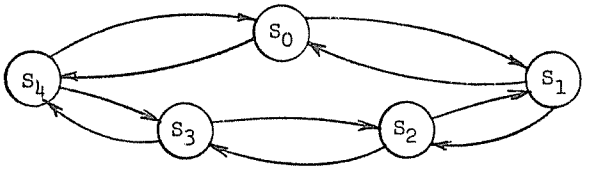
There are five classes, each containing one cyclic process (i.e. one "philosopher"), and ten exclusions :

If philosopher # i is eating then neither philosopher (i-1)mod 5 nor philosopher (i+1)mod 5 can begin to eat. (i = 0,1,2,3,4).

Eating is a critical sec+ion for each philosopher. We have :

$$S = \{S_0, S_1, S_2, S_3, S_4\} \qquad R = \{R_{i,(i-1)\bmod 5}, R_{i,(i+1)\bmod 5} \mid i = 0,..4\}$$

The graph of exclusions is :



The reduced solution, according to the second rule of equivalence is shown on Figure 1, and the Figure 2 gives the solution obtained by application of the first rule {Appendix} :

(philosopher # i)

do begin think ;

s_i:down $s_{(i-1)\bmod 5}$, $s_{(i+1)\bmod 5}$;

eat ;

up $s_{(i-1)\bmod 5}$, $s_{(i+1)\bmod 5}$

end

Fig. 1

(philosopher # i)

do begin think ;

$s_{(i-1)\bmod 5}$:$s_{(i+1)\bmod 5}$:down s_i ;

eat ;

up s_i

end

Fig. 2

Both programs may lead to different implementation but anyway, the net synchronizing effect is identical for both solutions. Indeed the exclusions are identical and therefore the rules of dependences are identical and the synchronizations are identical. This shows how the synchronizing primitives are only a representation of the intended synchronization.

5.5. The starvation of the philosophers.

It is well known that two philosophers (# i-1 and # i+1) can provoke the starvation of the philosopher between them (# i). If they are never at the same time in the "think" section of the program, philosopher #i can never enter its "eat" critical section. Any solution which forces any pair of non-adjacent philosopher to be together in the "think" section within a finite time after the philosopher between them is waiting to enter the "eat" critical section will be a solution to the starvation problem. It is sufficient that each philosopher, when waiting to enter the "eat" critical section, prevents the philosopher on his right to leave the "think" critical section.

To do this, we surround each "eat" section S_i by a section A_i and we define the following relation of exclusions :

When a philosopher # i is asking to eat, philosopher # (i+1)mod 5 cannot ask to eat.

The graph of exclusions is given on figure 3 and the program is on Figure 4 :

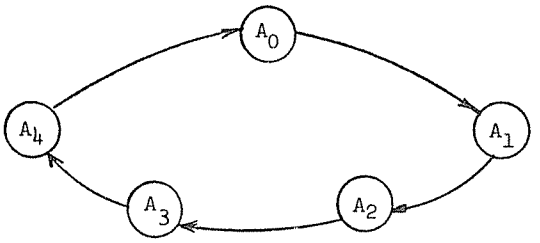


Fig. 3

```
(philosopher # i)
do begin  think ;
          a_i :down a_(i+1)mod 5 ;
          critical-section-S_i ;
          up a_(i+1)mod 5
end
```

Fig. 4

5.6. A finite buffer problem.

In this problem, there are two types of dependence rules. First, both classes of consumer and producer processes (the number of processes is not limited in this example)involve one critical section each : add to buffer and take from buffer. The critical sections of both classes are mutually exclusive :

S_a S_t

The relation of exclusions can be expressed by the d-operations as on Figure 5 :

```
s_ta:s_aa:down s_aa,s_at ;        s_at:s_tt:down s_ta,s_tt ;
        add-to-buffer ;                   take-from-buffer ;
        up s_aa,s_at                      up s_ta,s_tt
```

Fig. 5

```
s_b:down s_b ;          s_b:down s_b ;
add-to-buffer ;         take-from-buffer ;
up s_b                  up s_b
```

Fig. 6

Figure 6 shows the solution after reduction. It is obviously the P-V solution of the mutual exclusion problem.

Second, we have two rules of cooperation between the consumer processes and the producer processes. We assume that the producers (consumers) add (take) one portion to (from) the buffer for each execution of their critical section. Furthermore, we require that : 1. the number of portions taken from the buffer can never exceed the number of portions added to the buffer, augmented by the number q_f of initially filled places in the buffer, and 2. the number of portions added to the buffer can never exceed the number of portions taken from it, augmented by the number q_e of initially empty places. This gives the graph of cooperation of Figure 7.

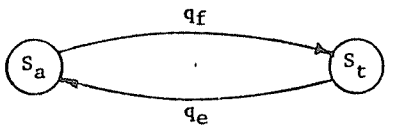


Fig. 7

s_e:down s_e;	s_f:down s_f ;
add-to-buffer ;	take-from-buffer ;
up s_f	up s_e

Fig. 8

We know that the initiation of S_t has a negative effect on its own possibility. This is programmed by using a semaphore as a closing operand and as a passage operand at the beginning of S_t. This gives the operation s_f:down s_f. The termination of S_a has a positive effect on the same possibility and therefore involves the operation up s_f. Since the number of initially filled portions is q_f, the semaphore s_f is initialized to q_f-1 . A similar construct is used to program the other cooperation. This gives the program on Figure 8.

The mutual exclusion must be applied to the same critical sections, and to achieve this, we combine the operations involved in the critical sections into one indivisible operation by means of the conjunction law. This gives the following program :

(producers)	(consumers)
s_b*s_e:down s_b,s_e ;	s_b*s_f:down s_b,s_f ;
add-to-buffer ;	take-from-buffer ;
up s_b,s_f	up s_b,s_e

Since we do not have an implementation for the conjunction law (cf. {6}), we can separate the exclusion and the cooperation as on Figure 9 or as on Figure 10.

(producers)	(consumers)
s_e:down s_e ;	s_f:down s_f ;
s_b:down s_b ;	s_b:down s_b ;
add-to-buffer ;	take-from-buffer ;
up s_b ;	up s_b ;
up s_f	up s_e

Fig. 9

(producers)	(consumers)
2	
s_b:down s_b ;	s_b:down s_b ;
	1
s_e:down s_e ;	s_f:down s_f ;
add-to-buffer ;	take-from-buffer ;
up s_f ;	up s_e ;
up s_b	up s_b

Fig. 10

The solution of Figure 9 is intuitively satisfactory and is often used in practice The solution of Figure 10 seems to be satisfactory as well but it contains an inconspicuous possibility of deadlock.

If the buffer is empty (i.e. s_f = -1) process *1* cannot proceed until process *2* adds something to the buffer. But *2* is prevented to do so because process *1* has set the value of s_b to -1 as well.

When one defines methods and tools for programming problems of dependences between parallel processes, the least one should require from such methods is that they do not introduce deadlocks into problems which are free of them at the level of the statenent of the problem. The method that we used to obtain the solution of this problem by means of the conjunction rule is satisfactory and verifies such requirenents. The way the programs of Figure 9 and 10 were obtained is unacceptable since the second program contains a deadlock whenever the original problem has none. However such programs are efficient and could be useful if one is able to detect when parasitic deadlocks occur (e.g. for simple programs).

6. IMPLEMENTATIONS.

6.1. Basic issues.

Implementations of synchronizing tools should be examined in the context of computing systems design (mostly operating systems and networks). The question could not be fully understood if it was presented merely as a problem of writing a piece of code that will "make the thing work" on a particular machine. Henceforth one cannot compare it with the problem of writing a compiler for a new language or of designing a software package for some particular application (e.g. graphics).

The implementation of synchronizing primitives has its own specific characteristics, for example the forms of waiting or the methods of queuing and releasing processes, but these aspects are subject to criteria of efficiency and applicability which could only be defined as design goals for a system as a whole. For example the forms of waiting can be classified in order of efficiency but this factor can only be appraised by defining the interactions between processes (loose or strong connections) and the usage of the resources (light or heavy load). Both of these factors are meaningful only as parameters of a whole system.

Basically, the implementation problems can be divided into three categories :

1. The semantic aspects : what to implement and what for ?
2. The scheduling problems : the algorithms for queuing and releasing processes.
3. The forms of waiting : active (busy) or passive (idle) waiting ?

Design decisions should be taken in function of two essential criteria :

1. The operational characteristics of the system : what should the processes be able to do, how do they use the resources, how sophisticated should be the synchronizing tools ?
2. The performance characteristics : available resources, timing constraints.

The semantic aspects are closely related to the problem of defining primitives as discussed in the preceding sections. It will therefore not be detailed here.

The scheduling aspects can be summarized as follows. When a process is waiting for an event possibility to become possible, the corresponding effect in the implementation will be to put on some queue the process representation (i.e. the data structure representing a process in the particular implementation). When the event possibility becomes possible again, the process should be released. This can be done in several ways, first in -first out, priority system, or simply chance. These decisions are properly decisions about the choice of an algorithm, made with the intention of guaranteeing some properties (e.g. it should avoid starvation). It is also known that some scheduling properties (e.g. priorities [14,49] and {5.3.}) can be enhanced by the processes themselves independently of any implementation. This is to say that synchronizing tools can be used for scheduling purposes. Restrictions on the built-in scheduling of the implementation can be achieved by a suitable programming of the processes. An interesting question is to study what kind of restrictions can be explicitly programmed and what kind should be included in the implementation. (About this cf. also {7.1.}). The example of implementation given further makes no definite decision about the scheduling. Actually the algorithm can be modified rather easily to fit different needs.

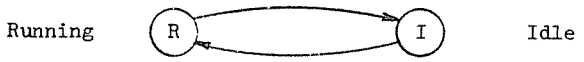
Only the problem of the waiting forms, out of the three aspects of implementation design, is discussed in details in the next section. It is indeed a contribution of this work to have proposed an implementation for a general primitives system without any form of busy waiting. Until recently, it was believed that this was possible only for very elementary(such as P-V) systems of synchronizing primitives.

6.2. The waiting forms.

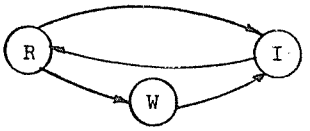
6.2.1. Some definitions.

The best way to give a clear definition of the waiting phenomenon is to assume the different processes to compete for a common resource called processor. This resource can exist in several units and each process will therefore compete for one of them.

The processes (which are theoretical entities) need a processor to realize their activity. At different instants of time a process is either associated with a processor (it is "running") or not (if no one is available ; the process is "idle"). We thus have for each process two states and the transitions between the states are governed by the competition for the resource.



Whenever a process attempts to execute a not-possible action, it is not allowed to proceed and it becomes"waiting" until the event becomes possible. We introduce a third state :



6.2.2. Busy form of waiting.

In the busy form of waiting, the W state is not distinguished from the R and I state ; i.e. a process can be "running" while "waiting"or "idle" while "waiting". This solution is known to be most unefficient because a process "running" while "waiting" retains a processor that could be used by another process.

6.2.4. Idle form of waiting.

In the idle form of waiting, no process can be in more than one state at a time ; i.e. if it is "waiting", it cannot be "running" or "idle". It accomplishes the transition R-W as soon as a not-possible event is encountered and the transition W-I as soon as this event becomes possible. The transitions R-I and I-R are resource competing transitions.

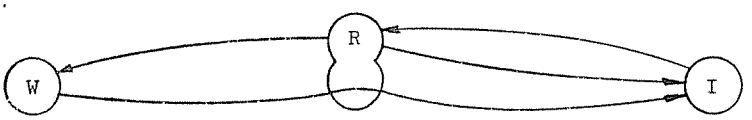
6.2.4. Controlled busy form of waiting.

If, for instance, process p is waiting for an event a to become possible, when this event becomes possible, p will go from state W to state I. But the event a did not occur yet (it can occur only during a "running"state for p) and it is possible that a becomes not-possible again due to some action of another process while p is in state I. When p goes eventually to R, it will be to attempt to execute a which is not-possible and p must go back to W. The whole cycle of p through I,R, and W was done for nothing.

This phenomenon should be avoided if one wants to keep the efficiency of the implementation. When it is deliberately left to chance, one has the controlled busy form of waiting. It is said to be controlled because the number of occurrences of such a phenomenon for one process p and one event a is bounded by the number of processes competing for the processors.

It is worthwhile to note that the claim of efficiency for the idle form of waiting is valid only within limits. If the activity of the processes is low, e.g. if there are few processes in the system, the possibility of events is almost always possible and there is a small probability to have processes in W state. In this case, it might be less costly to implement a busy form of waiting rather than a (time consuming) implementation of the switching between states. This remark was made by P. Brinch Hansen [5,6] .

A solution to the problem of the controlled busy form of waiting resides in a simple idea : no process should transit from W state to I state if the event that is now possible has not occurred. This gives the following transitions :



The process stops over in the R state to realize the action (it forces the event to occur) and then it goes to state I.

For some primitives, it is possible to deduce in principle that the event will remain possible under certain conditions while the process is in I state. Therefore no controlled busy form of waiting can occur if these conditions are verified at all time. This is used in the implementation of the P-V primitives.

6.3. The complexity of the implementation.

Another difficulty in optimizing implementations is that one should be able to calculate in an efficient way the new values of the possibilities after each event occurrence. This is not always an obvious task ; for instance with the d-operation :

$$p(\underline{x}) : \underline{x} \leftarrow f(\underline{x})$$

The variations of the value of $p(\underline{x})$ should be computed from the variations of the variables $\underline{x}$. The effective computation of all the p's would be much unefficient because they can be arbitrarily complex, and other means should be used to achieve these calculations.

Although the implementation presented here brings a solution to this problem in a particular case, no general solution is known. (About this problem see [43]).

6.4 Monitors.

In regard to the problem of implementation, a proposal of interest has been made with the concept of "monitors". ([3,25]).

A monitor is seen as a program component (module) that controls communications and synchronizations between parallel processes. It is defined by a set of restricted operations (accesses) on global data structures (resources) performed concurrently by processes. By restricted, it is meant that processes can only access the resources through the specified operations.

This concept of a modular well specified and protected "secretary" is presented as a reliable and efficient method to implement a variety of communication and allocation problems. In particular it is proposed as a tool for implementing synchronizing primitives and, somewhat beyond this operational use, it is argued to be able to replace the primitives themselves (cf. e.g. the problem of Readers-Writers in [25] and compare it with our solution {5.1.}).

It is somewhat suspicious that an implementation tool can suppress all the conceptual (semantic) problems related to synchronization. This confusion arises probably from another confusion between two levels of abstraction : the level of the primitives (i.e. in other words the level of the calls to the monitor services) and the level of the monitor itself (i.e. the implementation of the calls). The fact that the synchronizing primitives do not appear explicitly anymore at one level does not mean that the problem has vanished.

The implementation given further can be seen as a monitor that implements the d-operations. But we will at all time maintain a strict distinction between the levels of abstraction. This was taken as a foundation for all our research.

6.5. An implementation of the d-operations (with majority law).

We assume that all the operations implemented are d-operations with majority law and that they have only one single paasage operand. We have proven that it is always possible to transform any set of d-operations into another equivalent set where all the passage operands are single {Appendix}.

Two d-operations are different if their syntactic structures are different. The general notation used in this section is :

$$s_i : \underline{down}\ s_{j_1}, s_{j_2}, \ldots, s_{1_m}\ \underline{up}\ s_{k_1}, s_{k_2}, \ldots, s_{k_p}$$

A difference is made between a d-operation, an instance of a d-operation in the program text, and an execution of a d-operation. To any operation corresponds a data structure implementing a representation of the syntactic structure of the operation. To any instance of a d-operation corresponds one call of the procedure 'exec' that implements the synchronizing effects. To an execution of a d-operation corresponds an actual execution of the procedure 'exec'.

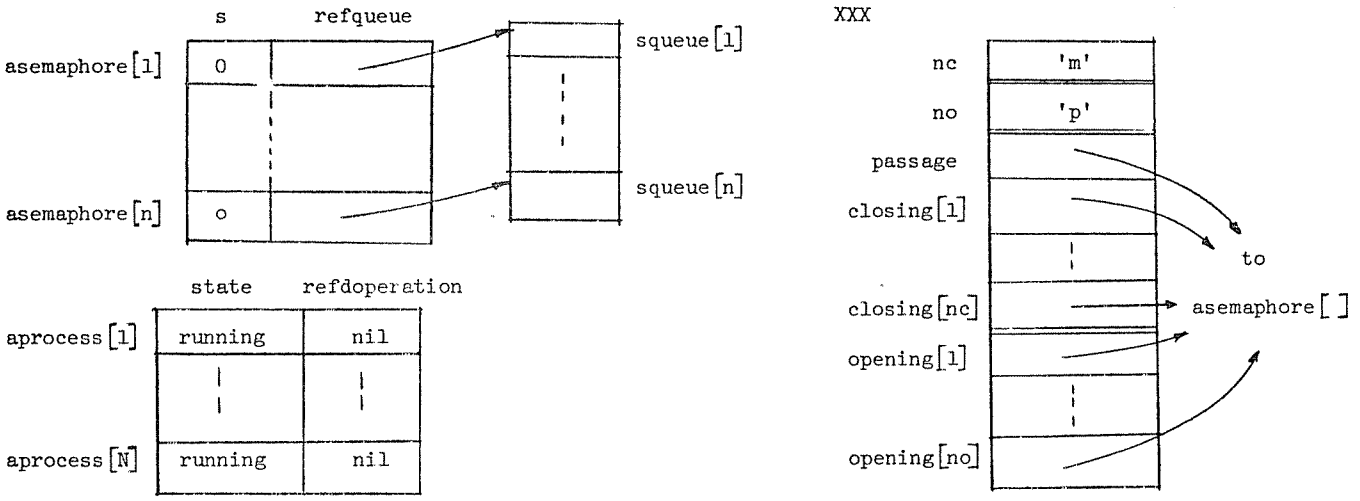
For simplicity we assume that the number of processes is N. The state of a process is either "running" or "waiting".

The program is written in PASCAL, except for the generation of the pointers variables for which we diverge from the Report. The method we use is self explanatory and avoids some complications.

```
{TYPE DEFINITIONS}

type semaphore = record s : integer ; refqueue : ↑queue end

type queue = sequence of  process {the type sequence is not a PASCAL type. It is introduced in [8]}

type process = record state : (running,waiting) ; refdoperation : ↑doperation end

type doperation = record nc, no : integer ; passage : ↑semaphore ;
                         closing , opening ; array[integer] of ↑semaphore end

{VARIABLE DECLARATION}

var asemaphore : array [1..n] of semaphore ; var squeue : array [1..n] of queue ;

var aprocess : array [1..N] of process ; var XXX : doperation {for every d-operation, a similar

declaration must be generated with a different identifier} ;

{VARIABLES INITIALIZATIONS}

for i:=1 to n do with asemaphore[i] do begin s:=o ; refqueue↑ := squeue[i] end ;

for i:=1 to N do with aprocess[i] do begin state := running ; refdoperation := nil end ;

{XXX = down s_j1,..,s_jm up s_k1,..,s_kp}
with XXX do begin nc := 'm' ; no := 'p' ; passage := nil ;
                  closing[1]↑ := asemaphore['j1'] ; ... ; closing[nc]↑ := asemaphore['jm'] ;
                  opening[1]↑ := asemaphore['k1'] ; ... ; opening[no]↑ := asemaphore['kp'] end

{XXX = s_i:down s_j1,..,s_jm up s_k1,..,s_kp}
with XXX do begin nc := 'm' ; no := 'p' ; passage↑ := asemaphore['i'];
                  closing[1]  := asemaphore['j1'] ; ... ; closing[nc]  := asemaphore['jm'] ;
                  opening[1]  := asemaphore['k1'] ; ... ; opening[no]  := asemaphore['kp'] end
```

{One of the two last initializations must be generated for each d-operation, and the identifiers between quotes must be replaced by the corresponding actual values.}

The following figure gives a picture of the data structure after the initializations.



Every time a process needs to execute a d-operation, it calls the procedure 'exec' and it gives as parameters its name and the name of the operation to execute.

The procedure is divided into four parts :

1. passage. The actual value of the passage operand is checked. If it exists and if it is negative, the process is put on the queue associated with the semaphore ; it becomes "waiting".
2. closing. The closing operands are decreased by one.
3. opening. The opening operands are increased by one.
4. wake up. If the values of the opening operands are positive, some processes must be waken up. When a process is waken up, the operation upon which it was "waiting" is executed. Then the program checks whether it is necessary to wake up some other processes or the semaphore value is negative again. Thus the procedure 'exec' is recursive.

The indivisibility of a call of 'exec' is ensured by the operating system. However this does not guarantee the indivisibility of the d-operations because the procedure 'exec' is recursive. Yet one can see that the indivisibility holds in this case. Indeed the parts 'passage'-'closing'-'opening' of the procedure are always executed together (i.e. without interleaving with any recursive call) and only the part 'wake up' can call 'exec' recursively. This shows that the different executions of 'passage'-'closing'-'opening' are done without any interaction between them, whatever is the degree of recursiveness. This is a sufficient condition for indivisibility.

```
{PROCEDURE FOR THE EXECUTION OF THE D-OPERATIONS}

procedure exec (var refprocess : †process ; var refdoperation : †doperation) ;
var pp : †process ;
begin with refdoperation† do
{passage} if (passage ≠ nil) ∧ (passage†.s < 0)
          then begin refprocess†.refdoperation := refdoperation ;
                     refprocess†.state := waiting ;
                     put(refprocess, passage†.refqueue†)
               end
          else begin if nc > 0
{closing}            then for i := 1 to nc do with closing[i]† do  s := s-1 ;
                     if no > 0
{opening}            then begin for i := 1 to no do with opening[i]† do  s := s+1 ;
{wake up}                       for i := 1 to no do with opening[i]† do
                                begin get(pp, refqueue†) ;
                                      pp†.state := running
                                      exec(pp, pp†.refdoperation) ;
                                      pp†.refdoperation := nil
                                end
                          end
               end
end
```

7. DEADLOCKS.

7.1. Introduction.

7.1.1. Origin of the problem.

From an historical point of view, deadlocks were found within operating systems in a empirical manner. They were due to the lack of coordination between processes competing for reusable but of exclusive access resources. They were believed to be errors of operating systems and algorithms were designed to avoid these errors. Most of these algorithms were scheduling and allocation algorithms for the accesses to the resources.

7.1.2. A problem of duality.

It is well known that competing for resources can be interpreted as the synchronization of the executions of critical sections delimiting the accesses to these resources : e.g. owning a resource in a exclusive way (e.g. a magnetic tape) is equivalent to executing in a exclusive way the critical sections where all the accesses (e.g. read and write) to the resource are performed.
A deadlock prevention algorithm can thus be seen as a scheduling of the allocation of resources or as a scheduling within the implementation of synchronizing primitives.

As it was pointed out earlier, it is difficult to differentiate the scheduling aspects and the pure synchronizing effects of a system of synchronizing primitives (cf. e.g. the priority problem {5.3.} and the starvation{5.5}). And very often, a given scheduling requirement can be programmed by synchronization or a synchronization problem can be implemented by scheduling [33] .

Because of this simple observation, we can reasonably state the following assertions :

1. Given a particular deadlock prevention scheduling, it is always possible to design a program whose deadlock problem will not be solved by the algorithm.

2. Given a particular scheduling that does not prevent deadlocks (except in trivial cases), it is always possible to increase the synchronization requirements of any program to make it deadlock free (i.e. given a scheduling, it is possible to define the class of deadlock free programs).

Neither of these questions has been investigated seriously and, a fortiori, neither has been proven true.

7.1.3. The game of deadlock.

All the studies, as far as we know, on deadlocks were oriented toward the design of prevention algorithms for the allocation of reusable resources (e.g. [10,11,17,21,23,27,28,38,48]).
One of the best ways to summarize the different approaches has been introduced by Devillers [17] : it is an interpretation in terms of the theory of games.

Within such an interpretation, the two players of the game are :

1. The scheduling algorithm (SA) which detains all the resources and whose moves in the game are to allocate these resources to processes.

2. A group of processes which use the resources and whose moves in the game are to ask for resources or to release resources they have been allocated by prior moves of the SA.

The beginning of the game is usually defined by an initial state where the SA has all the available resources and the processes have none of them. The first move is therefore for the processes to make a request for some resources. The succession of possible moves defines the states of the game, each state giving the move to either the processes or the SA.

If the number of processes is finite and if so is the number of their moves, then there is at least one state where there is no move to be made by the processes anymore. If the game reaches this state, it is a winning game for the SA (i.e. the deadlocks were prevented).

If the game does not reach a final state, we have a winning game for the processes. For reasons not developed here (cf. [17]), this can happen only if there is a state which defines impossible moves for the SA (i.e. in which a move cannot be made without allocating more resources than the number available). Such a state is a completely blocked state.

If between a given state and the final state there are no other paths than paths going through one or several completely blocked states then this given state is a deadlock.

The strategy of the SA will be to keep the game in states where it is always possible to find a path to the final (winning !) state. Such states are called safe states.

We can now characterize the different approaches to the problem in a dual way {7.1.2.} :

1. Assuming a strategy of the processes (i.e. the (future) history of their resources utilization), what is a winning strategy for the SA ? The strategy of the processes can be specified in different ways : - by a maximum need of resources during the whole game [21] , or - by a complete history (task step model) [17,23] with or without possibility of abortion. These are the problems most widely studied in the literature.

2. Given a strategy of the SA(a scheduling algorithm which can be trivial e.g. pick up any move at random), what is the class of winning strategies for the processes ? This is the object of our work.

In an infinite game (i.e. if the number of processes and/or their number of moves are infinite) there is a possibility of an infinite number of states(although there are possible definitions of a finite number of final states). In this case a game that will not reach the final state is a winning game for the processes. This can be due to a deadlock (as in the finite case), to an effective deadlock [26], or to starvation.

7.1.4. The present work.

Our work will deal with a special case of game where the number of processes is infinite. They are however divided into classes of processes ; all processes of the same class have only a finite number of possible moves. A necessary and sufficient condition is given, that specifies the winning strategies for the processes. The problem of effective deadlock is not investigated ; it is a possible extension to this research.

The strategy of the SA is assumed to be based on the following principle :

If a process requests a resource then the SA acknowledges the request within a finite bounded time.†

The winning strategies of the processes are represented by relations of exclusions between critical sections. The exclusions can be seen as equivalent to requests of accesses to re-usable resources. Each graph of exclusions will specify rules of accesses to one resource of one type. These conditions are altogether less general (there is only one resource of each type) and less particular (accesses are more general than exclusive accesses) than the classical problem studied in the literature.

The theorems given here define a constructive way of defining a priori whether a given system of processes can generate deadlocks.

7.1.5. Deadlocks and Correctness.

The problem of the characterization of deadlocks is a typical semantic problem of parallel processes programming. It can be compared to a correctness problem in sequential programming (e.g. does a program terminate ?). It falls within the scope of this research as an example of the possibilities of a sound theoretical model for defining and proving properties of programs. The class of problems of deadlocks studied here is limited but non trivial, and of a wide application in practice.

---

† Obviously, this condition is verified only if the game is not in a completely blocked state. It is a strong condition that, in particular, avoids effective deadlocks and starvation.

7.2. The theorems : the general case.

7.2.1. Basic definitions.

The framework of definitions that we need to proceed to the demonstration of the theorems were, for the essential,introduced earlier. They are briefly recalled hereafter.

DEFINITION 1.

* A set of programs is defined by a set A of integer variables called places. Each place corresponds to one operation of the program and to one action of the processes executing it.

  a^i_j , $1 \le i \le N$, $1 \le j \le N_i$, $\pi_i = a^i_1 \dots a^i_{N_i}$

* The relation $a^i_j \le a^i_k$ holds between two places if $j \le k$

DEFINITION 2.

* An interval I^i_{jk} is defined in the following way : $I^i_{jk} = [a^i_j, a^i_k] = \{a^i_\ell \mid j \le \ell \le k\}$
* $\overline{I^i_{jk}} = a^i_j$, $\underline{I^i_{jk}} = a^i_k$
* $[a^i_j]$ is a one place interval noted a^i_j

DEFINITION 3.

* A state s is a set of values for the a^i_j : $[a^i_j]_s \ge 0$, $1 \le i \le N$ $1 \le j \le N_i$
* $\pi_s = \sum_{i,j} [a^i_j]_s$ is the population of the state s .
* The population of an interval is : $[I^i_{jk}]_s = \sum_{\ell} [a^i_\ell]_s$ $j \le \ell \le k$
* A state s' is a sub-state of a state s if : for all i,j $[a^i_j]_{s'} \le [a^i_j]_s$

DEFINITION 4.

* A transition τ^i_j , $1 \le j \le n_i+1$, holds between two states s_1 and s_2 in one of the followimg cases :
  * If $j=1$ then $[a^i_j]_{s_2} = [a^i_j]_{s_1} + 1$ and $[a^\ell_k]_{s_2} = [a^\ell_k]_{s_1}$ for all $(\ell,k) \neq (i,j)$
  * If $j=N_i+1$ then $[a^i_j]_{s_2} = [a^i_j]_{s_1} - 1$ and $[a^\ell_k]_{s_2} = [a^\ell_k]_{s_1}$ for all $(\ell,k) \neq (i,j)$
  * If $1 < j < N_i$ then $[a^\ell_k]_{s_2} = [a^\ell_k]_{s_1}$ for all $(\ell,k) \neq (i,j)$ and all $(\ell,k) \neq (i,j-1)$

    $[a^i_j]_{s_2} = [a^i_j]_{s_1} + 1$ and $[a^i_{j-1}]_{s_2} = [a^i_{j-1}]_{s_1} - 1$

Intuitively, the value attached to a place in a given state represents the number of processes present in this place (i.e. having executed the operation a^i_j but not yet the following one). The definition of a transition specifies formally the fact of going from one place to the next one, and the fact of entering the system (an initial transition) and of leaving the system.

DEFINITION 5.

* C is defined as a set of intervals called critical sections.
* GE is a relation on C called the relation of exclusions.
* Valid notations for GE are :

  A graph notation (the graph of exclusions).

  If $C_1, C_2 \in C$, $(C_1, C_2) \in GE$ is equivalent to $C_1 \Xi C_2$ (C_1 excludes C_2).

DEFINITION 6.

* A transition τ_j^i is <u>admissible</u> in state s if either $j = 1$ or $[a_{j-1}^i]_s > 0$

* A transition τ_j^i is <u>not-possible</u> in state s if there are two critical sections C_1 and C_2 such that : 1. $C_1 \Xi C_2$, 2. $[C_1]_s > 0$, and 3. $\overline{C_2} = a_j^i$. Otherwise it is <u>possible</u>.

* A transition is <u>enabled</u> if it is both admissible and possible.

DEFINITION 7.

* A <u>derivation</u> is a sequence of states : $s_1 s_2 s_3 \ldots\ldots s_k$ such that :

for all j, $1 < j \leq k$, σ_j is an enabled transition in the state s_{j-i} and it puts the system in state s_j . We use the notations $s_{j-1} \overset{\sigma_j}{\rightarrow} s_j$ and $s_1 \overset{*}{\rightarrow} s_k$.

DEFINITION 8.

* The <u>initial state</u> of a system of processes is noted α and is defined by : $\pi_\alpha = 0$

* A state s is <u>realizable</u> if : $\alpha \overset{*}{\rightarrow} s$ for some derivation.

These definitions specify under which conditions a system can go from one state to another one. This is done by moving a process from one place to another (a transition). In order to do that, we must be sure that the place the process is coming from is "inhabited" (the transition is admissible), and that no holding exclusion forbids this transition (the transition is possible).

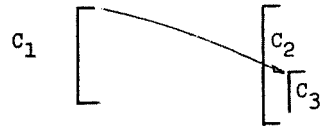
The system is assumed to start from an initial state (when there is no process in the system). The only states that can be reached from the initial state are called realizable states.

DEFINITION 9.

* The <u>critical graph</u> CG is defined on the set C of critical sections :

(C_1, C_2) is an directed edge of CG if there is a third critical section C_3 such that 1. $C_1 \Xi C_3$, and 2. if $C_2 = I_{j_2 k_2}^{i_2}$ and $C_3 = I_{j_3 k_3}^{i_3}$ we have $i_3 = i_2$ <u>and</u> $j_2 < j_3 \leq k_2 + 1$

This is a formal (and unfortunately complicated) way of defining a simple situation described in the following figure :



The beginning of C_3 cannot be the beginning of C_2 but it can be any other place including the one right after the end of C_2.

7.2.2. The single-process postulate.

We give now a property of the system that we will assume to hold in all the systems that we shall study. This postulate expresses that a system which contains at any instant exactly one process has only realizable states. This is equivalent to say that no single process requests more resources than available. We then give a necessary and sufficient condition for this postulate to be satisfied.

POSTULATE. (Single-process postulate).

* All states s such that $\pi_s = 1$ are realizable. *

THEOREM 1. (Necessary and sufficient condition (N.S.C.) for the single-process postulate).

* A N.S.C. for the single-process postulate is that the critical graph CG be loop-free.*

We define the hypothesis H to be the single-process postulate and the thesis T to be the absence of loops in the critical graph. The proof is as follows :

$\forall$ 1. $H \rightarrow T$. We prove "ad absurdum" i.e. we prove $\sim T \wedge H \rightarrow \sim H$.
Assume that the critical section C is a loop of the graph CG ; we certainly have by definition of CG (Def. 9) a critical section C' such that :

1. $C \equiv C'$, and 2. if $C = I^i_{jk}$ and $C' = I^i_{j'k'}$ then $j < j' \leq k'+1$

It is easy to see that no state s such that $[C']_s = 1$ and $\pi_s = 1$ is realizable. Indeed there must be a state s' such that $\pi_{s'}=1$ and $[a^i_{j'-1}]_{s'}=1$ in order to obtain the state s . But the transition $\tau^i_{j'}$ is never enabled in s'. Hence s is not realizable. $\forall$

$\forall$ 2. $T \rightarrow H$. T implies that in each state s , $\pi_s = 1$ and $[a^i_j]_s = 1$, the transition τ^i_{j+1} is enabled.

Then we have at least one derivation from the initial state to any such s state :

$$\alpha \xrightarrow{\tau^i_1} s_1 \xrightarrow{\tau^i_2} s_2 \longrightarrow \cdots\cdots \longrightarrow s_{j-1} \xrightarrow{\tau^i_j} s \quad .$$

7.2.3. Incompatible places.

In this section, we define a property of a set of places. Then we prove a theorem that gives necessary and sufficient conditions for that property. This property is a key concept of our study of the deadlocks problems. Indeed a system of processes will be deadlock free if the statea of that system that are deadlocks are non realizable states. These states are divided into classes, each class corresponding to a set of places. If the states are realizable then there will be at least one process occupying each of the places. If they are not realizable this cannot occur and we say that the set of places is incompatible.

Actually, the incompatibility will be defined on instances of places (not on places). Instances of places can have values in a state s of the system. The value of the place is the sum of the values of its instances. Intuitively, we identify a process being in a place for some state s, as an instance of this place with the value 1 in s. These concepts are purely formal but they are much helpful in the proof of the theorems.

DEFINITION 10.

* A set of instances of places $\{\hat{a}^{i1}_{\ell 1}, \hat{a}^{i2}_{\ell 2}, \cdots\cdots, \hat{a}^{in}_{\ell n}\}^\dagger$ is <u>incompatible</u> if there is no realizable state s such that : for every j, $1 \leq j \leq n$ $\quad [\hat{a}^{ij}_{\ell j}]_s > 0$

Note that, because we deal with instances of places, it is possible that for some j and k :

$$\hat{a}^{ij}_{\ell j} \ .=. \ \hat{a}^{ik}_{\ell k} \ \dagger\dagger$$

DEFINITION 11.

* The notation $a^{i1}_{j1} \equiv a^{i2}_{j2}$ is for : There are two critical sections C_1 and C_2 such that

$$((a^{i1}_{j1} \subseteq C_1) \wedge (a^{i2}_{j2} = \overline{C}_2) \wedge (C_1 \equiv C_2))$$

i.e. a^{i1}_{j1} is in a critical section that "closes the door" a^{i2}_{j2} . The inclusion is defined as usually between intervals,and a place is a single place interval.

---

† We use the circumflex to indicate that these are instances.

†† We use the equality .=. to avoid confusions. The two symbols must be equal, not their values.

LEMMA 1. (Sub-states lemma).

* Every sub-state s' of a realizable state s is realizable. *

∇. Let s be a realizable state. We have for some derivation : $\alpha \xrightarrow{\tau^{i_1}_{j_1}} s_1 \xrightarrow{\tau^{i_2}_{j_2}} \cdots\cdots \xrightarrow{\tau^{i_n}_{j_n}} s_n = s$

We define : $\alpha' \xrightarrow{\sigma_1} s'_2 \xrightarrow{\sigma_2} \cdots\cdots \xrightarrow{\sigma_n} s'_n = s'$

where : $\sigma_k = \tau^{i_k}_{j_k}$ if $[a^{i_k}_{j_k}]_{s'_k} > 0$; $\sigma_k = \varepsilon$ (no transition) otherwise .

We first prove the following : If s'_k is a sub state of s_k then s'_{k-1} is a sub-state of s_{k-1} .

First, if $\sigma_k = \varepsilon$ we consider the three possible cases :

$(i,j) \neq (i_k,j_k)$ and $\neq (i_k,j_k-1)$ $\quad [a^i_j]_{s'_{k-1}} = [a^i_j]_{s'_k} \leq [a^i_j]_{s_k} = [a^i_j]_{s_{k-1}}$

$(i,j) = (i_k,j_k)$ $\quad [a^i_j]_{s'_{k-1}} = [a^i_j]_{s'_k} = 0 \leq [a^i_j]_{s_{k-1}}$

$(i,j) = (i_k,j_k-1)$ $\quad [a^i_j]_{s'_{k-1}} = [a^i_j]_{s'_k} \leq [a^i_j]_{s_k} = [a^i_j]_{s_{k-1}} - 1 < [a^i_j]_{s_{k-1}}$

Second, if $\sigma_k = \tau^{i_k}_{j_k}$:

$(i,j) \neq (i_k,j_k)$ and $\neq (i_k,j_k-1)$ $\quad [a^i_j]_{s'_{k-1}} = [a^i_j]_{s'_k} \leq [a^i_j]_{s_k} = [a^i_j]_{s_{k-1}}$

$(i,j) = (i_k,j_k)$ $\quad [a^i_j]_{s'_{k-1}} = [a^i_j]_{s'_k} - 1 \leq [a^i_j]_{s_k} - 1 = [a^i_j]_{s_{k-1}}$

$(i,j) = (i_k,j_k-1)$ $\quad [a^i_j]_{s'_{k-1}} = [a^i_j]_{s'_k} + 1 \leq [a^i_j]_{s_k} + 1 = [a^i_j]_{s_{k-1}}$

Hence, since s' is a sub-state of s , α' is a sub-state of α and $\alpha' = \alpha$.∇

THEOREM 2.

* A N.S.C. for a set of instances of places :

$$\{\hat{a}^{i_1}_{j_1}, \hat{a}^{i_2}_{j_2}, \cdots\cdots, \hat{a}^{i_n}_{j_n}\} \quad , \quad \underline{n > 1}$$

to be incompatible is : For every u, $1 \leq u \leq n$ one of the following case holds :

H1. There is a v, $1 \leq v \leq n$, $v \neq u$ such that $a^{i_v}_{j_v} \equiv a^{i_u}_{j_u}$ †

H2. If $j_u - 1 \neq 0$, $\{\hat{a}^{i_1}_{j_1}, .., \hat{a}^{i_u}_{j_u-1}, \cdots, \hat{a}^{i_n}_{j_n}\}$ is incompatible ††

H3. If $j_u - 1 = 0$ and if $n > 2$ $\{\hat{a}^{i_1}_{j_1}, .., \hat{a}^{i}_{j\,(u-1)\,(u-1)}, \hat{a}^{i}_{j\,(u+1)\,(u+1)}, \cdots, \hat{a}^{i_n}_{j_n}\}$ is incompatible †††

∇1. Sufficient condition : $\forall u(H1(u) \vee H1(u) \vee H3(u)) \rightarrow T$. We proceed "ad absurdum" and we prove :

$$\sim T \wedge \forall u(H1(u) \vee H2(u) \vee H3(u)) \rightarrow \exists u(\sim H1(u) \wedge \sim H2(u) \wedge \sim H3(u))$$

Because we have T (and by application of the sub-states lemma) there is a state s such that :

$$\text{for every } u,\ 1 \leq u \leq n\ [\hat{a}^{i_u}_{j_u}]_s = 1 \quad \text{and} \quad \pi_s = n$$

Therefore there is a state s' and a v, $1 \leq v \leq n$ such that :

$$\text{for every } u,\ 1 \leq u \leq n,\ u \neq v\ [\hat{a}^{i_u}_{j_u}]_{s'} = 1 \quad \text{and} \quad [\hat{a}^{i_v}_{j_v-1}]_{s'} = 1 \text{ and } \pi_{s'} = n$$

This because s is realizable and there must be a last process to "enter" s from s'.
But the transition from s' to s certainly contradicts H1 , and s' itself contradicts H2 and (if applicable) H3.

---

† Removing the circumflex represents the place whose instance was represented by the symbol with the circumflex.

†† i.e. an instance of the place preceding the place whose instance is $\hat{a}^{i_u}_{j_u}$, replaces $\hat{a}^{i_u}_{j_u}$ in the set.

††† There is no third case if $n = 2$. It would lead to an absurdity because if $\{a^i_j\}$ is incompatible, it contradicts the single-process postulate.

$\nabla 2$. Necessary condition. $T \rightarrow \forall u(H1(u) \vee H2(u) \vee H3(u))$ We proceed "ad absurdum" and we prove :

$$T \wedge \exists u(\neg H1(u) \wedge \neg H2(u) \wedge \neg H3(u)) \rightarrow \neg T$$

If H3 is not applicable this gives us a state s and a u such that :

$$\text{for every } v,\ 1 \leq v \leq n,\ v \neq u \quad \left[\hat{a}^{i_v}_{j_v}\right]_s = 1 \quad \text{and} \quad \left[\hat{a}^{i_u}_{j_{u-1}}\right]_s = 1 \quad \text{and} \quad \pi_s = n$$

But we have $H1(u)$ and therefore the transition $\tau^{i_u}_{j_u}$ is enabled in s and produces a state s' such that : for every v, $1 \leq v \leq n$ $\left[\hat{a}^{i_v}_{j_v}\right]_{s'} = 1$ and $\pi_{s'} = n$ and $s \xrightarrow{*} s'$

This contradicts T. If H3 is applicable the proof is similar. If n=2 and if H3 is applicable, we obtain the state s by application of the single process postulate, and then the proof is similar to the other cases.∇

Remark. If we want to use this theorem to prove on a concrete example that some set of instances is incompatible, we need to apply the theorem recursively. Because the number of places is finite and because the number of instances cannot increase from one application to another, the theorem has a "fixed point", i.e. the example will be proven after a finite number of recursive applications of the theorem. Ultimately, by application of the theorem, the existence of some incompatibility is reduced to the existence of exclusions between places, This reduces a dynamic property (dependent on the evolution of the system) to a static property (dependent on the stucture of the processes and on the exclusions).

7.2.4. The definitions of deadlocks.

In this section we give definitions of deadlocks and of completely blocked states together with equivalence properties between them. Intuitively, a completely blocked state is (if we speak now in terms of resources) a state where all processes are in demand of resources while no resources are available. In our model, it means a state where no transition is enabled. A deadlock is a state for which there is no derivation leading to other states than completely blocked states.

DEFINITION 12.

* A completely blocked state s is a realizable state such that :

1. $s \neq \alpha$, and 2. no non-initial transitions are enabled in s .

DEFINITION 13.

* A deadlock is a realizable state s for which there is no derivation such that :

$$s \xrightarrow{*} \alpha \qquad (\alpha \text{ is the initial but also the final state.})$$

We now prove a preliminary theorem. This theorem is important because it states the equivalence between an infinite class of derivations and a finite one.

THEOREM 3.

* For any realizable state s , if there is a derivation δ : $s \xrightarrow{*} \alpha$ then there is a derivation δ' without initial transition such that : $s \xrightarrow[\delta']{*} \alpha$.*

∇. We have by hypothesis :

$$s_0 = s \xrightarrow{\tau^{i_1}_{j_1}} s_1 \xrightarrow{\tau^{i_2}_{j_2}} s_2 \text{ - - - - } \xrightarrow{\tau^{i_n}_{j_n}} \alpha$$

We define :

$$s'_0 = s \xrightarrow{\sigma_1} s'_1 \xrightarrow{\sigma_2} s'_2 \text{ - - - - } \xrightarrow{\sigma_n} \alpha'$$

where : 1. $\sigma_k = \varepsilon$ if $\tau^{i_k}_{j_k}$ is an initial transition (i.e. $j_k = 1$)

2. $\sigma_k = \varepsilon$ if $\left[a^{i_k}_{j_k-1}\right]_{s_{(k-1)}} = 0$ and 3. $\sigma_k = \tau^{i_k}_{j_k}$ otherwise.

We prove now that if s'_{k-1} is a sub-state of s_{k-1} then s'_k is a sub-state of s_k .

There are three cases to consider :

1. $\tau_{j_k}^{i_k}$ is an initial transition :

for all $(i,j) \neq (i_k,1)$ $\quad [a_j^i]_{s'_k} = [a_j^i]_{s'_{k-1}} \leq [a_j^i]_{s_{k-1}} = [a_j^i]_{s_k}$

for $(i,j) = (i_k,1)$ $\quad [a_1^{i_k}]_{s'_k} = [a_1^{i_k}]_{s'_{k-1}} \leq [a_1^{i_k}]_{s_{k-1}} = [a_1^{i_k}]_{s_k} - 1 \leq [a_1^{i_k}]_{s_k}$

2. $|a_{j_k}^{i_k}|_{s_{k-1}} = 0$: $(i,j) = (i_k,j_k)$ $\quad [a_j^i]_{s'_k} = [a_j^i]_{s'_{k-1}} \leq [a_j^i]_{s_{k-1}} \leq [a_j^i]_{s_{k-1}} + 1 = [a_j^i]_{s_k}$

$(i,j) = (i_k,j_k-1)$ $\quad [a_j^i]_{s'_k} = [a_j^i]_{s'_{k-1}} = 0 \leq [a_j^i]_{s_{k-1}} - 1 = [a_j^i]_{s_k}$

for all $(i,j) \neq (i_k,j_k)$ and $\neq (i_k,j_k-1)$

$$[a_j^i]_{s'_k} = [a_j^i]_{s'_{k-1}} \leq [a_j^i]_{s_{k-1}} = [a_j^i]_{s_k}$$

3. $\sigma_k = \tau_{j_k}^{i_k}$: $(i,j) = (i_k,j_k)$ $\quad [a_j^i]_{s'_k} = [a_j^i]_{s'_{k-1}} + 1 \leq [a_j^i]_{s_{k-1}} + 1 = [a_j^i]_{s_k}$

$(i,i) = (i_k,j_k-1)$ $\quad [a_j^i]_{s'_k} = [a_j^i]_{s'_{k-1}} - 1 \leq [a_j^i]_{s_{k-1}} - 1 = [a_j^i]_{s_k}$

for all $(i,j) \neq (i_k,j_k)$ $\neq (i_k,j_k-1)$ $\quad [a_j^i]_{s'_k} = [a_j^i]_{s'_{k-1}} \leq [a_j^i]_{s_{k-1}} = [a_j^i]_{s_k}$

This shows that α' is a sub-state of α and thus $\alpha' = \alpha$.∇

We now prove that the existence of a deadlock is equivalent to the existence of a completely blocked state.

LEMMA 2.

* A completely blocked state is a deadlock. *

∇. Trivial.∇

LEMMA 3.

* If there is a deadlock in a system then there is a completely blocked state in that system. *

∇. We proceed "ad absurdum" . Assume s is a deadlock. Since there is no completely blocked state there is at least one non initial transition enabled in s,that leads to a state s_1 .

By a similar argument we conclude that there is a potentially infinite derivation of non initial transitions : $s \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} s_2$

But the finiteness of the system implies that for some n , bounded, $s_n = \alpha$. Hence s is not a deadlock.∇

THEOREM 4.

* In a system, there is no deadlock if and only if there is no completely blocked state. *

∇. Trivial by application of Lemma 2 and Lemma 3 ∇

7.2.5. Theorems of existence.

We establish a relation between the existence of completely blocked states and the incompatibility of some set of instances of places in the system. This sets the missing link between the existence of deadlocks and the existence of some structural properties of the relations of exclusions.

DEFINITION 14.

* A critical embrace corresponding to a cycle of the critical graph CG

$$C_1, C_2, C_3, \ldots, C_n$$

is a set of instances of places : $\{\hat{a}^{i_1}_{j_1}, \hat{a}^{i_2}_{j_2}, \ldots, \hat{a}^{i_n}_{j_n}\}$ such that :

for all ℓ, $1 \leq \ell \leq n$, $C_\ell = I^{i}_{p_\ell q_\ell}$, there is a critical section K and the following holds ;

$$a^{i}_{j_\ell+1} = \overline{K} \quad \text{and} \quad C_{\ell-1} \Xi K \quad \text{and} \quad p_\ell < j_\ell+1 \leq q_\ell+1 \qquad (\text{if } \ell=1 \text{ then } \ell-1 = n)$$

By definition of the critical graph there is at least one critical embrace for each cycle of CG.
The intuitive notion of the critical embrace is close to the notion of critical graph :
For each cycle of the graph each critical section of the cycle "closes" something inside the next one.
There is thus a possibility for processes executing this cycle to block each other. If this happens, they will be blocked in a critical embrace.

LEMMA 4.

* A set of instances of places $\{\hat{a}^{i_1}_{j_1}, \hat{a}^{i_2}_{j_2}, \ldots, \hat{a}^{i_n}_{j_n}\}$ such that for each k, $1 \leq k < n$, $a^{i_k}_{j_k} \Xi a^{i}_{j_{(k+1)}+1}$ and $a^{i_n}_{j_n} \Xi a^{i}_{j_1+1}$, is a critical embrace. *

∇. Proof by application of the definition.∇

We prove now the fundamental theorem of existence.

THEOREM 5.

* In a system, there is no completely blocked state if and only if all critical embraces are incompatible. *

We define H as the incompatibility of the critical embraces and T as the absence of completely blocked states.

1. $H \rightarrow T$. We proceed "ad absurdum" : $\sim T \wedge H \rightarrow \sim H$.

Assume that s is a completely blocked state. Let $\{a^{i_1}_{j_1}, a^{i_2}_{j_2}, \ldots, a^{i_m}_{j_m}\}$ be the set of places such that $\pi_s = \sum_k [a^{i_k}_{j_k}]_s$ and for all k, $1 \leq k \leq m$, $[a^{i_k}_{j_k}]_s > 0$, i.e. there is no processes anywhere else than in these places. None of the following transitions is possible :

$$\tau^{i_1}_{j_1+1} \quad \tau^{i_1}_{j_2+1} \quad \ldots\ldots \quad \tau^{i_m}_{j_m+1}$$

This implies that for all k, $1 \leq k \leq m$, there is a u, $1 \leq u \leq m$, $u \neq k$ such that $a^{i_u}_{j_u} \Xi a^{i_k}_{j_k+1}$
If we now represent these exclusions as a graph whose nodes are the $a^{i_k}_{j_k}$ and where $(a^{i_u}_{j_u}, a^{i_v}_{j_v})$ is an (directed) edge if $a^{i_u}_{j_u} \Xi a^{i_v}_{j_v+1}$, then this graph is cyclic. (This is a property of directed graphs. If x is a node of a graph X and if for every x there is an edge going from one node of $X - \{x\}$ to x, then the graph is cyclic.)
Hence, by lemma 4, there is a critical embrace : $\{a^{i_{u_1}}_{j_{u_1}}, a^{i_{u_2}}_{j_{u_2}}, \ldots, a^{i_{u_n}}_{j_{u_n}}\}$ † $1 \leq u_k \leq m$, $1 \leq k \leq n$ such that for each u_k $[a^{i_{u_k}}_{j_{u_k}}]_s > 0$. This contradicts H.

† A set of places is a set of instances of places.

V2. $T \to H$. We proceed "ad absurdum" : $T \wedge \sim H \to \sim T$.

If there is a non incompatible critical embrace $\{\hat{a}^{i}_{j_1}, \hat{a}^{i_2}_{j_2}, \ldots, \hat{a}^{i_n}_{j_n}\}$

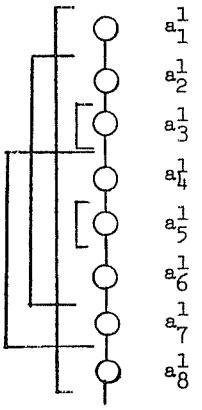
then by the sub-state lemma there is a state s such that :

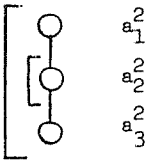
$$\pi_s = n \quad \text{and for all } k,\ 1 \leq k \leq n, \quad \left[a^{i_k}_{j_k}\right]_s = 1$$

In s no non initial transitions are possible.

7.3. Example.

We treat an example that illustrates all the theorems proven in Section 7.2.. We have two classes of processes :





The critical sections :

$A_1 = I^1_{18}$ $\qquad B_1 = I^2_{13}$

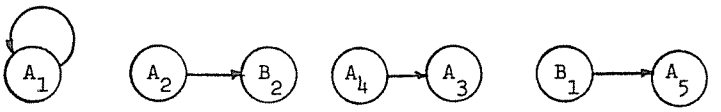
$A_2 = I^1_{26}$ $\qquad B_2 = a^2_2$

$A_3 = a^1_3$

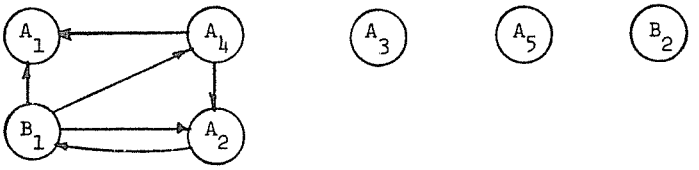
$A_4 = I^1_{47}$

$A_5 = I^1_5$

The graph of exclusions :



The critical graph :



The cycles of the critical graph : (A_2, B_1) $\qquad (A_2, B_1, A_4)$

The critical embraces : $\{a^1_4, a^2_1\}$ $\qquad \{a^1_2, a^2_1, a^1_4\}$

It is obvious that $\{a^1_4, a^2_1\}$ is not incompatible because $a^1_4 \not\equiv a^2_1$. Thus there is a completely blocked state. It is realized by the derivation : $\tau^1_1\, \tau^1_2\, \tau^1_3\, \tau^1_4\, \tau^2_1$

We prove for illustration that $\{a^1_2, a^2_1, a^1_4\}$ is incompatible.

To do so we have to prove that the following ones are incompatible :

$\{a^1_1, a^2_1, a^1_4\}$ $\qquad \{a^1_2, a^1_4\}$ $\qquad \{a^1_2, a^2_1, a^1_3\}$

These are reduced to the following by application of the theorem (and because $a^1_4 \equiv a^1_1$) :

$\{a^1_1, a^1_4\}$ $\quad \{a^1_1, a^2_1, a^1_3\}$ $\quad \{a^1_2, a^1_3\}$ $\quad \{a^1_2, a^2_1, a^1_2\}$

By new applications of the theorem on the preceding ones and because $a_4^1 \equiv a_1^1$ and $a_3^1 \equiv a_1^1$, we have to prove the following ones to be incompatible :

$\{a_1^1, a_3^1\}$ $\{a_1^1, a_1^2, a_2^1\}$ $\{a_1^1, a_2^1\}$ $\{a_2^1, a_2^1\}$

Then, because $a_3^1 \equiv a_1^1$ and $a_2^1 \equiv a_1^1$, we have to prove that $\{a_1^1, a_1^1\}$ and $\{a_1^1, a_1^2, a_1^1\}$ are incompatible. This reduces to the proof of $\{a_1^1, a_1^1\}$ to be incompatible, which is true because $a_1^1 \equiv a_1^1$. Hence $\{a_2^1, a_1^2, a_4^1\}$ is incompatible.

7.4. The theorems for nested critical sections.

When one restricts the structures of the processes and accepts only nested critical sections, the proofs of absence of deadlocks are simplified in a dramatic manner. The aim of this section is to provide the needed theorems to achieve these results. The principles of the proofs are similar to the ones encountered earlier.

DEFINITION 15.

* Let $I_{j_1k_1}^{i_1}$ and $I_{j_2k_2}^{i_2}$ be two intervals, we write :

$I_{j_1k_1}^{i_1} \subset I_{j_2k_2}^{i_2}$ if $j_2 < j_1 \leq k_1 < k_2$ and $I_{j_1k_1}^{i_1} \subseteq I_{j_2}^{i_2} k_2$ if $j_2 \leq j_1 \leq k_1 \leq k_2$

DEFINITION 16.

* A set of critical sections defines nested critical sections if for any pair C_1, C_2 of critical sections : $((C_1 \subset C_2) \vee (C_2 \subset C_1) \vee (C_1 \cap C_2 = \emptyset))$

DEFINITION 17.

* Let $C_1, C_2, \ldots, C_n$ be a cycle of the critical graph CG and

$$\begin{matrix} \lambda_1^1, \lambda_2^1, \ldots, \lambda_n^1 \\ \lambda_1^2, \lambda_1^2, \ldots, \lambda_n^2 \\ \cdot \quad \cdot \quad \cdot \\ \lambda_1^{m_1}, \lambda_2^{m_2}, \ldots, \lambda_n^{m_n} \end{matrix}$$ Its corresponding critical embraces.

We define for each k, $1 \leq k \leq n$ $\quad I_k = \left[\overline{C_k}, \max_{\leq}(\lambda_k^1, \lambda_k^2, \ldots, \lambda_k^{m_k})\right]$

The set of instances of intervals $\{I_1, I_2, I_3, \ldots, I_n\}$ is called the critical embracing intervals of the cycle of CG .

THEOREM 6.

* In a system with nested critical sections, there is no completely blocked states if and only if the critical embracing intervals of each cycle of CG are incompatible .*

∇1. The sufficient condition is proven in the same way as the sufficient condition of Theorem 5 was proven.

∇2. Necessary condition. $T \rightarrow H$. We proceed "ad absurdum" : $\sim H \wedge T \rightarrow \sim T$

We assume that there is a non incompatible set of critical embracing intervals :

$I = \{I_1, \ldots, I_2\}$ $\quad I_k = I_{j_k \ell_k}^{i_k}$

There is a state s in which $\pi_s = n$ and for each k, $1 \leq k \leq n$, $\quad [I_k]_s = 1$

The definition of the λ_j^i, $1 \leq j \leq n$, $1 \leq i \leq m_j$ implies that the only transitions that are not-possible inside the I_k are the ones corresponding to the λ_j^i that is :

if $\lambda^i_j = a^u_v$ then τ^u_{v+1} is not possible. Let assume that in I_k, $[a^{i_k}_{u_k}]_s = 1$ $(j_k \leq u_k \leq \ell_k)$ then $\tau^{i_k}_{u_k+1}$ is enabled and from s the system can go to a state s'. This is possible iteratively until $a^{i_k}_{u_k+v_k} = \lambda^k_w$ for some w, $1 \leq w \leq m_k$. This derivation is possible for each I_k. Starting with I_1, we can now write a derivation :

$$s \xrightarrow{*} s_1 \xrightarrow{*} s_2 \xrightarrow{*} \qquad \xrightarrow{*} s_n$$

and s_n is a completely blocked state. Hence $\sim T$.∇

We need to give now the theorems giving necessary and sufficient conditions for a set of instances of intervals to be incompatible. But we need first a (last !) definition.

DEFINITION 18.

* Let $I = \{I_1, \ldots, I_n\}$ be a set of instances of intervals, $I_u = I^{i_u}_{j_u k_u}$ $n > 1$

We define for each u,v, $u \neq v$:

$H(I_u, I_v)$ = (there is a critical section C_w and a place $a^{i_v}_{\ell_v}$ such that

$$((I_u \subseteq C_w) \wedge (\ell_v \leq j_v) \wedge (C_w \Xi a^{i_v}_{\ell_v}))$$

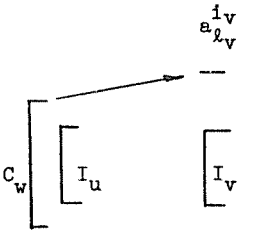


Fig. 1

This defines the Figure 1.

We now consider the set of instances of places such that: For all u, $u \neq v$, $H(I_u, I_v)$ is true and we call ω_u the maximum of this set for the relation $\leq$. We define $K_u = [\omega_u . a^{i_u}_{k_u}]$, $1 \leq u \leq n$. If ω_u does not exist then K_u is <u>not defined</u> .

THEOREM 7.†

* In a system with nested critical sections, a N.S.C. for a set of instances of intervals to be incompatible is : for all u, $1 \leq u \leq n$ one of the following cases holds :

H1. if K_u is defined then $K_u = I_u$

H2. if K_u is defined then $\{I_1, \ldots, I_{u-1}, K_u, I_{u+1}, \ldots, I_n\}$ is incompatible.

H3. if K_u is not defined and if $n > 2$ then $\{I_1, \ldots, I_{u-1}, I_{u+1}, \ldots U_n\}$ is incompatible. *

$\nabla 1$. The sufficient condition is trivial.∇

$\nabla 2$. Necessary condition. $T \rightarrow \forall u(H1(u) \vee H2(u) \vee H3(u))$. We proceed "ad absurdum" and we prove :

$\exists u(\sim H1(u) \wedge \sim H2(u) \wedge \sim H3(u) \wedge T \rightarrow \sim T$

If K_u is defined and if $n > 2$ then there is a state such that : $\pi_s = n$ and for all v, $v \neq u$ $[I_v]_s = 1$ and $[K_u]_s = 1$.

---

† We keep the same notations as in Definition 18.

It is obvious that no transitions are not-possible in $K_u - I_u$ and therefore there is a derivation from s to a state s' where :

$$\text{For all } u,\ 1 \le u \le n \qquad [I_u]_{s'} = 1$$

If K_u is not defined then there is a state s where for all v, $v \neq u$, $[I_v]_s = 1$. $\pi_s = n-1$

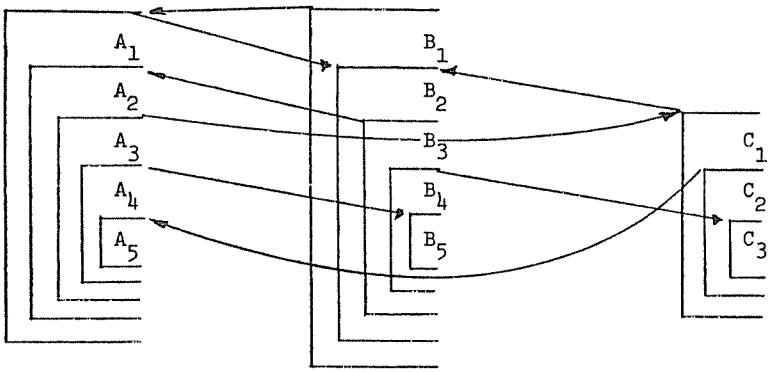
Since there is no not-possible transitions between $a_1^{i_u}$ and $a_{j_u}^{i_u}$ (including the initial transition), otherwise K_u would be defined, there is a derivation from s to a state s' where

for all u, $1 \le u \le n$, $[I_u]_{s'} = 1$ $\pi_{s'} = n$

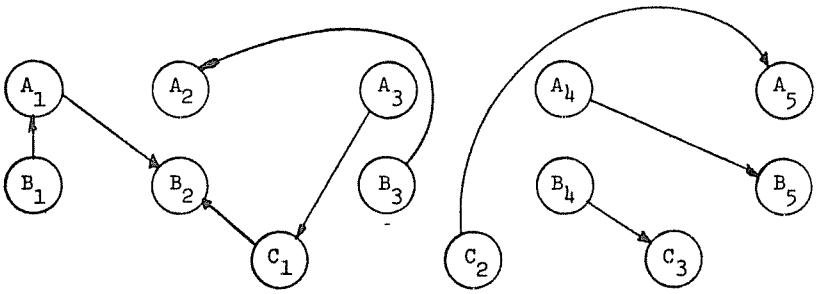
If n = 2 then K_u must be defined. By using the single process postulate we find easily a state that contradicts T .∇

7.5. Example with nested critical sections.

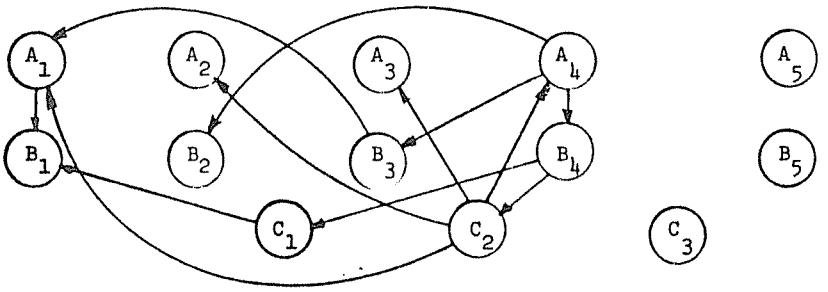
The structure of the processes is :



The graph of exclusions :



The critical graph :



The cycles of the critical graph are : (A_4, B_4, C_2)

The critical embracing intervals and the intervals defined upon them by definition 18. :

$I_1 = [\overline{A}_4, \overline{A}_5]$ $\quad I_2 = [\overline{B}_4, \overline{B}_5]$ $\quad I_3 = [\overline{C}_2, \overline{C}_3]$

$K_1^1 = [\overline{A}_2, \overline{A}_5]$ $\quad K_2^1 = [\overline{B}_2, \overline{B}_5]$ $\quad K_3^1 = [\overline{C}_1, \overline{C}_3]$

$K_1^2 = [\overline{A}_1, \overline{A}_5]$ $\quad K_2^2 = K_2^2$ $\quad K_3^2$ is not defined

$K_1^3 = K_1^2$

We have to prove that $\{I_1, I_2, I_3\}$ is incompatible.

This gives the following ones to be incompatible by application of the theorem :

$\{I_1, K_2^1, I_3\}$ $\quad \{I_1, I_2, K_3^1\}$ $\quad \{K_1^1, I_2, I_3\}$

Then we obtain :

$\{K_1^1, K_2^1, I_3\}$ $\quad \{I_1, K_2^1, K_3^1\}$ $\quad \{K_1^1, I_2, K_3^1\}$ $\quad \{I_1, K_2^1, K_3^1\}$

$\{I_1, I_2\}$ $\quad \{K_1^2, I_2, I_3\}$

And :

$\{K_1^2, K_2^1, I_3\}$ $\quad \{K_1^1, K_2^1, K_3^1\}$ $\quad \{I_1, K_2^1\}$ $\quad \{K_1^2, I_2, K_3^1\}$

$\{K_1^1, I_2\}$

And :

$\{K_1^2, K_2^1, K_3^1\}$ $\quad \{K_1^1, K_2^1\}$ $\quad \{K_1^2, I_2\}$

And :

$\{K_1^2, K_2^1\}$

The last set is incompatible because $K_1^3 = K_1^2$ and $K_2^2 = K_2^1$. Thus this example has no deadlocks.

8. CONCLUSIONS.

If we had to conclude these pages by one phrase, we could appropriately write :
By using a semantic method whose foundation is a distinction between the level of the formal models, the level of the programming tools, and the level of the implementation practicalities, we have made possible a synthesis of the problems of synchronization.

The method is of a semantic nature because it attempts to represent intuitive concepts and to fit practical situations. It attains to a synthesis because it encompasses a large variety of problems and because it provides means of comparing them. It also separates the theoretical issues from the practical problems. Ideally, facing a problem of synchronization, one should first devise an appropriate theoretical abstraction, and then proceed by examining the needed programming tools and by tackling the problems of implementation. With this method, a given problem is characterized by its statement, preferably formal, and its transcription into some programming language and into some implementation can be done almost automatically.

One of the most helpful theoretical constructs of this thesis was to distinguish classes of problems of synchronization (e.g. exclusions, cooperations) and to define formal rules to characterize them (the rules of dependence). Different classes are defined upon a common root of basic concepts and hypotheses. In this way, we were able to show how analog problems can be handled in an analog way and how different problems can be fundamentally different.

The traditional emphasis on the definition of new synchronizing primitives becomes a secondary preoccupation. It is now much akin to the definition of programming languages : it has its proper issues (e.g. syntax, error detection) but it does not suffice to solve general questions.

The main unanswered question of this thesis resides in a comprehensive understanding of the implementation. Partial results have improved the current state of the art but one fundamental question remains : what is an implementation ? Or more exactly : how should one define the interface between the implementation and the abstract semantic level ? This is definitely a promising area for future research.

Another unanswered question is the duality between synchronization and scheduling. The way events occur when they become possible is specified only by the "within a finite time" rule. For some problems other specifications are needed (e.g. priorities, scheduling in general). It is therefore conceivable to define a d-operation without scheduling (the order in which the processes are freed is "immaterial") and a d-operation with scheduling. What are the implications of this question is also an interesting area of research.

The problem of deadlocks arose as a finger exercise on a given theme : the exclusions between critical sections. The results obtained are important as they provide strong theorems on the problem. Furthermore, they raise a deeper question : are deadlocks only pathological behaviours of the allocation of resources, or are they intrinsic characteristics of concurrent programs. The research pursued for this thesis suggests that part of the truth lies in the second alternative : problems of synchronization can generate deadlocks whatever is the way they are programmed.

Finally the conclusion of these conclusions is : Beyond the many answers that we have found along our route, either in the works of other researchers or in our own efforts, there are still many unknown areas, and at the end of these pages we cannot but write, modestly, a question mark.

9. APPENDIX.

It is evident that different primitives can describe the same synchronizing effect. This was illustrated on the examples of section 5. In some sense, certain structures of primitives can be defined equivalent if they program the same synchronization. The purpose of this Appendix is to give a formal treatment of these equivalences for the d-operations with majority law.

The two theorems given here can be used to transform a set of d-operations into an equivalent set where all the passage (resp. closing and opening) operands are single. The equivalences are defined as linear transformations on the semaphore variables. In the definitions, one must handle carefully all the particular cases, and this obscures the simplicity of the method.

If we take the d-operations with majority law, their general form can be defined :

$p(\underline{x}) : \underline{x} \leftarrow f(\underline{x})$ $\qquad p(\underline{x}) = \sum_i a^i x_i \geq 0 \qquad a^i \in \{0,1\}$ (i is an index for a^i)

$f(\underline{x}) = \underline{x} - \underline{d} + \underline{u}$

$\underline{x} = \begin{bmatrix} x_1 \\ . \\ . \\ x_n \end{bmatrix} \qquad \underline{d} = \begin{bmatrix} d_1 \\ . \\ . \\ d_n \end{bmatrix} \qquad \underline{u} = \begin{bmatrix} u_1 \\ . \\ . \\ u_n \end{bmatrix}$ the d_i and the u_i are positive integers.

We can define the a_i by the vector $\overline{a} = [a_1 \; . \; . \; a_n]$ and $p(\underline{x}) = \overline{a}\underline{x} \geq 0$

We can now define N_p passage conditions :

$$p_j(\underline{x}) = \sum_i a^i_j x_i \qquad 1 \leq j \leq N_p$$

by the matrix $P = \begin{bmatrix} \overline{a}_1 \\ \overline{a}_2 \\ .. \\ . \\ \overline{a}_{N_p} \end{bmatrix}$ of order (N_p,n)

and similarly, N_d closing operands vectors and N_u opening operands vectors by the matrices :

D of order (n,N_d) and U of order (n,N_u)

A set S of d-operations defined on P, D, and U is represented by a set of N triples T :

$(i, j, k) \qquad 0 \leq i \leq N_p,\ 0 \leq j \leq N_d,\ 0 \leq k \leq N_u$

The operation (i, j, k) is written :

$$\overline{p}_i\underline{x} \leq 0 : \underline{x} \leftarrow \underline{x} - \underline{d}^j + \underline{u}^k$$

where $\overline{p}_i$ is the i-th row of P, $\underline{d}^j$ the j-th column of D, and $\underline{u}^k$ the k-th column of U. If one index is zero (say i = 0), the corresponding operand is absent (in this case the passage operand).

Definition. * Two sets of d-operations :

$S = \langle \underline{x}, n, N, N_p, N_d, N_u, P, D, U, T \rangle$ and $S' = \langle \underline{y}, m, M, M_p. M_d, M_u, Q, E, V. R \rangle$

are linearly equivalent if :

1. $M = N$, $M_p = .N_p$, $M_d = N_d$, $M_u = N_p$, and $R = T$

2. There are linear transformations : $Lx = y$ and/or $x = Ry$

such that : $P = QL$ and/or $Q = PR$
$E = LD$ and/or $D = ER$
$V = LU$ and/or $U = VR$ *

This means that two sets of d-operations are linearly equivalent if the variables x and y are in linear relation by R and L (including their initial values) and if the net effect of synchronization is identical since :

$$PR\underline{y} = P\underline{x} = Q\underline{y} = QL\underline{x}$$

Theorem 1.

$S = \langle \underline{x}, n, N, N_p, N_d, N_u, P, D, U, T \rangle$ and $S' = \langle \underline{y}, N_p, N, N_p, N_d, N_u, Q, E, T \rangle$

where $Q = 1_{N_p}$ are linearly equivalent with : $L = P$, $E = PD$, $V = PV$

Proof. Immediate since $P = QL = QP = 1_{N_p} P = P$

This means that S is equivalent to a set of d-operations S' where all the passage operands are single passage operands.

Theorem 2.

$S = \langle \underline{x}, n, N, N_p, N_d, N_u, P, D, U, T \rangle$ and $S' = \langle \underline{y}, N_d, N, N_p, N_d, N_u, Q, E, V, T \rangle$

where $E = 1_{N_d}$, $U = VR$, $Q = PR$, $R = D$, are linearly equivalent.

Proof. Immediate since $D = ER = ED = 1_{N_d} D = D$

This means that S is equivalent to S' where there are only single closing operands.

Remarks.

1. From a constructive point of view, if we assume P, D, U given, E and Q can be computed immediately, and V is an arbitrary matrix (with positive integer elements) such that $U = VD$

2. If $U = D$ (and $N_d = N_u$) then : $S' = \langle \underline{y}, N_d, N, N_p, N_d, N_d, Q, 1_{N_d}, 1_{N_d}, T \rangle$
In S' all closing and opening operands are single.

As an example of application of the second theorem, we can take the first problem of Readers-Writers {5.1.}. The programs were :

s_{wr}:down s_{rw} ;
reading ;
up s_{rw} ;

s_{rw}:s_{ww}:down s_{wr},s_{ww}
writing ;
up s_{wr},s_{ww} ;

and

s_{ww}:down s_{rw} ;
reading ;
up s_{rw} ;

s_{ww}:s_{rw}:down s_{ww} ;
writing ;
up s_{ww} ;

For the first program, the set of d-operations is defined by (using the notations of theorem 2) :

$$\underline{x} = \begin{bmatrix} s_{wr} \\ s_{rw} \\ s_{ww} \end{bmatrix} \qquad n = 3 \qquad P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \qquad N_p = 2 \qquad D = U = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad N_u = N_d = 2$$

$$T = \{(1,1,0),\ (0,0,1),\ (2,2,0),\ (0,0,2)\}$$

For the second program, we have :

$$\underline{y} = \begin{bmatrix} s_{rw} \\ s_{ww} \end{bmatrix} \qquad m = N_d = 2 \qquad E = V = 1_{N_d} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad M_d = M_u = N_d = N_u = 2$$

$$Q = PD = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \qquad R = T$$

This gives, for the second program, the operations :

s_{ww}:<u>down</u> s_{rw} <u>up</u> s_{rw} s_{rw}:s_{ww}:<u>down</u> s_{ww} <u>up</u> s_{ww}

REFERENCES.

1. Belpaire, G.,Wilmotte, J.P. A Semantic Approach to the Theory of Parallel Processes. Proc. of the International Computing Symposium 73, A. Günther (Ed.), North-Holland (1974) 159-164.

2. Belpaire, G., Wilmotte, J.P. Semantic Aspects of Parallel Processes. ACM SIGPLAN/SIGOPS Interface Meeting, SIGPLAN Notices 8, 9, (September 1973) 41-45.

3. Brinch Hansen, P. Concurrent Pascal : A Programming Language for Operating Systems Design. Information Sciences Tech. Report 10. (April 74), California Institute of Technology.

4. Brinch Hansen, P. Deamy : A Structured Operating System. Information Sciences Tech. Report. (May 74), California Institute of Technology.

5. Brinch Hansen, P. A Comparison of two Synchronizing Concepts. Acta Informatica 1, 190-199 (1972).

6. Brinch Hansen, P. A Reply to "Comments on "A Comparison of Two Synchronizing Concepts"". Acta Informatica 2, 189-190. (1973).

7. Brinch Hansen, P. Structured Multiprogramming. Comm. ACM 15, 574-578. (1972).

8. Brinch Hansen, P. Operating Systems Principles. Prentice -Hall, Englewood Cliffs, NJ. (1973)

9. Brinch Hansen, P. The Nucleus of A Multiprogramming System. Comm. ACM 13, 4 (April 1970) 238-241, 250.

10. Coffman Jr., E.G., Elphick, M.J., and Shoshani, A. System Deadlocks. Computing Surveys 3, 2 (1971), 67-78.

11. Coffman Jr., E.G., and Denning, P.J. Operating Systems Theory. Prentice-Hall, Englewood Cliffs, NJ, (1973)

12. Courtois, P.J., Heymans, F., and Parnas, D.L. Comments on "A Comparison of Two Synchronizing Concepts". Acta Informatica 1, 375-376 (1972).

13. Courtois, P.J., Heymans, F., and Parnas, D.L. Concurrent Control with "Readers" and "Writers". Comm. ACM 14, 10, (October 1971), 667-668.

14. Courtois, P.J., and van Lamsweerde, A. Some Extensions of the Concurrent Control with "Readers" and "Writers". MBLE Research Lab. Report R162 (March 71), Brussels.

15. Cerf V.G., Multiprocessors, Semaphores, and a Graph Model of Computation. University of California Report UCLA-10P14-110 (UCLA-ENG-7223) (1972), Los Angeles.

16. Dennis, J.B., and Van Horn, E.C. Programming Semantics for Multiprogrammed Computations. Comm. ACM 13, 4 (March 66), 143-155.

17. Devillers, R. Préventions des Interblocages dans un Modèle a Organigrammes. Thèse, Université Libre de Bruxelles (1974).

18. Dijkstra, E.W. Cooperating Sequential Processes. In Programming Languages, F. Genuys (Ed.), Academic Press (1968).

19. Dijkstra, E.W. The Structure of the T.H.E. Multiprogramming System. Comm. ACM 11, 5 (May 68) 341-346.

20. Dijkstra, E.W. Hierarchical Ordering of Sequential Processes. Acta Informatica 1, 115-138 (1971).

21. Habermann, A.N. Prevention of System Deadlocks. Comm. ACM 12, 7 (1969), 373-377, 385.

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

23. Hebalkar, P.G. A Graph Model for Analysis of Deadlock Prevention In Systems with Parallel Computations. Proc. IFIP Congress 1971, Vol. TA-3, 168-172.

24. Hoare, C.A.R., and Wirth, N. An Axiomatic Definition of the Programming Language Pascal. Acta Informatica 2, 335-355 (1973).

25. Hoare, C.A.R. Monitors : An Operating Systems Structuring Concept. Report STAN-CS-73-401, Stanford University (1973).

26. Holt, A.W, and Commoner,FF. Events and Conditions. Record of the Project MAC Conference on Concurrent Systems and Parallel Computations, ACM (December 1970), 3-52.

27. Holt, R.C. On Deadlocks in Computer Systems. Tech. Report CSRG-6 (1971), University of Toronto.

28. Holt, R.C. Some Deadlock Properties of Computer Systems. Computing Surveys 4, 179-196 (1972).

29. Holt, R.C. Comments on "Prevention of System Deadlocks". Comm. ACM 14, 1 (1971), 36-38.

30. Keller, R.M. Parallel Program Schemata and Maximum Parallelism :
1. Fundamental Results, Journal ACM 20, 3 (July 1973), 514-537.
2. Construction of Closures. Journal ACM 20, 4 (October 1973), 696-710.

31. Karp, R.,and Miller, R.E. Parallel Program Schemata. Journal of Computer and Systems Sciences 3, 147-195 (1969).

32. Lipton, R.C. On Synchronization Primitives Systems. Tech. Report of the Dept. of Computer Science, Carnegie-Mellon University (1973), and Research Report of the Dept. of Computer Science, Yale University (1973).

33. Lipton, R.C. Schedulers as Enforces on Synchronization Processes. Colloques IRIA, Aspects Théoriques et Pratiques des Systèmes d'Exploitation (1974).

34. Parnas, D.L. Information Distribution Aspects of Design Methodology. Proc. IFIP Congress 71 (1971).

35. Parnas, D.L. A Technique for Specification of Software Modules with Examples. Comm. ACM 15, 5 (May 1972), 330-336.

36. Parnas, D.L. On the Criteria to Be Used in Decomposing Systems into Modules. Comm. ACM 15, 12 (December 1972), 1053-1058.

37. Parnas, D.L. On a "Buzzword" : Hierarchical Structure. Proc. IFIP Congress 74, Vol. 2, 336-339 (1974).

38. Parnas, D.L., and Habermann, A.N. Comments on Deadlock Prevention Methods. Comm. ACM 15, 9 (1972), 840-841.

39. Patil, S.S. Coordination of Asynchronous Events. Project MAC Report MAC-TR 72 (June 1970), Massachusetts Institute of Technology.

40. Price, W.R. Implication of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems. Ph.D Thesis, Carnegie-Mellon University (June 73).

41. Shoshani, A.,and Bernstein, A.J. Synchronization in Parallel Data Bases. Comm. ACM 12, 11 (1969), 604-607.

42. Slutz, D.R. The Flow Graph Schemata Model for Parallel Computation. Project MAC Report MAC-TR 53 (September 68), Massachusetts Institute of Technology.

43. Sintzoff, M, and van Lamsweerde, A. Constructing Correct and Efficient Concurrent Programs. MBLE Research Lab. Report R266 (1974), Brussels.

44. Spier M.J., Hastings, T.N., and Cutler, D.N. An Experimental Implementation of the Kernel/Domain Architecture.

45. Spier, M. The Experimental Implementation of a Comprehensive Intermodule Communication Facility. Proc. 73 Sagamore Conference on Parallel Processes (1973), Syracuse University, Syracuse, NY.

46. Spier, M. Process Communication Prerequisites or the IPC-Setup Revisited. Ibidem.

47. Thomas, R.H. A Model for Process Representation and Synthesis. Project MAC report MAC-TR 83, Massachusetts Institute of Technology.

48. van Lamsweerde, A. Deadlock Prevention in Real Time Systems. Proc. International Computing Symposium 73, A. Günther (Ed.), North-Holland Pub. Co. (1973), 135-142.

49. Vantilborgh, H., and van Lambsweerde, A. On an Extension of Dijkstra Semaphore Primitives. Information Processing Letters 1, 5 (October 72), 181-186.

50. van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., and Koster, C.H.A. Report on the Algorithmic Language ALGOL 68. Numerische Mathematik 14, 79-218 (1969).

51. Wodon, P.L. Still Another Tool for Controlling Cooperating Algorithms. Report of the Department of Computer Science, Carnegie-Mellon University (1972).

52. Campbell, R.H. The specification of process synchronization by path expressions. Colloques IRIA, Aspects Théoriques et Pratiques des Systèmes d'Exploitation (1974).