Contents Chapter Four

CHAPTER FOUR

THE SECTION MODEL

4.1. INTRODUCTION

We begin this chapter with a fundamental discussion of coordination problems in general. The simple exclusion and sequencing problems which may occur in a multiprocessing system are visualized by the construction of an analogon. The analogon consists of the suggestive every-day scene of people entering and leaving rooms via one or more doors. The coordination problems may then occur in the passing of doors (assuming that only one person can pass a door at a time) and within the rooms (the purposes of different visitors in a room may be conflicting).
The discussion yields new insights into the nature and causes of coordination problems. Based on these insights we will develop a new descriptive method for process coordination problems, which we call the 'section model'.

The model to be developed here is inevitably based on previous work of many others. For instance, the view that coordination problems and solutions can best be described in the form of abstract relations (or *dependences*) between program parts or processes was expressed earlier by among others **Belpaire '75.** (See the overview of Belpaire's model in section 2.2.5, page 96.)
The use of state invariants and abstract gate-variables was examined earlier by **Robinson & Holt '73.** (See the overview of their model in section 2.3.1, page 101.)
The fundamental ideas of *abstraction* and *stepwise refinement* have been elaborated by Dijkstra and others. (See the overview in chapter 3.)
We have borrowed and combined several important ideas from these and other authors, without however denying ourselves the pleasure of adjusting and extending them freely to our present views and research goals.

In two appendices to this chapter we will elaborate some extensions: we will study the formalization of alternative control-flow structures (appendix C), and we will study the possibilities for an automized analysis of coordination structures, with the aid of a transition table model (appendix D).

4.2. COORDINATION PRINCIPLES

4.2.1. The analogon of the door[1]

Occasionally two people trying to enter a room through the same door bump into each other. Though the chances of 'bumping into somebody' are rather low, as experience shows, we may well ask ourselves the question of whether conflicts at a doorway can in principle be avoided completely by standard decision making strategies. Two aspects must be considered in detail then: simultaneity and equality.

Simultaneity
Clearly, all conflicts could be avoided if we would be able to assign unambiguous priorities to all people who will ever pass through the door which is being considered. Usually the person who arrives at a door first is given the highest priority in passing (though exceptions are thinkable). There would then be no problem if only one person can arrive at the doorway at the time. Actually, simultaneity is the real problem: who will go first, and who will go after, when two or more people arrive at the doorway at indistinguishable moments. 'Politeness' is no solution, at least not if everybody is equally polite. But then, neither is 'impertinency'. In the first case people may find themselves entangled in a perpetual "after you", "no, after you" block, in the second case a conflict implies a clash.
Fortunately, two people are almost never equally polite, or equally impertinent, so a conflict is not very likely.

Equality
Now, knowing that a conflict is not very likely does not mean that it will never occur. Let us consider therefore the case of two or more equally polite people in more detail. How can these people resolve their conflicts? They must decide which of them will enter the room first. If all people are indeed 'equal' in all respects, the problem is unsolvable. When there is no selection criterion there can be no selection. The obvious rebuttal against this argument is that 'when there is no selection criterion' *any* selection will do. We may just choose one of the persons in conflict, at random and allow him to enter the room first. But that is no real solution of the problem studied here. *Which* person will make the random choice? There can clearly be only one such person, as a random selection performed twice or more will almost certainly yield conflicting results. Instead of choosing the person which is allowed to enter the room first, the people in conflict must now choose the person which will make that choice for them (we will call such a person a *referee*). We have only moved the problem to another level, not solved it. We are still left with the problem that 'equal people cannot resolve their mutual conflicts'.
This observation need however not disturb us. Distinguishable things are by definition never completely equal, and non-distinguishable things may well be considered as one. This implies that if we cannot distinguish between two people in any respect, we must consider 'them' as a single person. (Note that, to be completely 'equal' two people must occupy precisely the same space, etc.) In this discussion we can therefore disregard complete equality.
Note also that (hypothetical) complete equality can cause no conflict: two people doing indistinguishable things in indistinguishable places at precisely the same time, are not in conflict at all.

Knowing this, we can phrase the 'problem of the doorway' more strictly: can we devise a refereeing process which guarantees that each person from a group of

— — — — —

(1) A similar analogon was discussed by **Atkinson & Hewitt '75**: the front-hall and reception desk of a large hospital.

N persons (N larger than one), which have all arrived at the doorway simultaneously, has an equal and non-zero chance to pass the doorway as the n^{th} person, for all $1 \leqslant n \leqslant N$.

Clearly, when inequality is used to determine access-priorities *directly* the people will not have equal chances in the above sense. Some people will always have to wait for others. One method then is to alter the access-priorities from time to time. The restrictions are however that:

(1) two persons may never obtain the same priority;
(2) all competitors know each others access-priority;
(3) the correct arbitration at the doorway is never confused by the changes in the priorities.

We have discussed such a strategy in appendix A to chapter 1 (**Algorithms for concurrent programming control**). Still, there are some alternative approaches. We will discuss two of these alternatives below, which we call the method of the 'floating referee' and the method of the 'doorkeeper'.

The floating referee
Instead of using the inequalities between competitors directly to assign access-priorities we may use them to choose a *referee*. When N persons arrive at a doorway simultaneously they will first choose a referee and the referee will then devise an ordering (at random) of the competitors, but without favoring or disfavoring himself. The task of the referee 'floats' and we will therefore speak of the method of the floating referee.
The restrictions are:

(1) no two persons may have the same refereeing priority;
(2) no two persons may obtain the same entrance priority;
(3) all persons know each other's refereeing priority;
(4) the referee will not favor or disfavor himself.

The doorkeeper
With the method of the floating referee we need two selections:

(1) the selection of a referee, and
(2) the selection by the referee.

We can omit the first selection if we use a fixed referee, which we call a 'doorkeeper'. The refereeing is then performed by an 'outsider', who is not himself involved in the conflict as such. The doorkeeper can be a person (process) or a machine (circuit), or even a combination of these. The doorkeeper uses the inequality between the competitors to distinguish and arbitrate between them, while guaranteeing equal access chances for all. The method of the doorkeeper is simple and easy to understand. On the other hand, the method of the floating referee is self-containing: people passing the doorway do not depend on the perpetual presence and correct service of an 'outsider'.

4.2.2. The guarded room

Up to this point we have only considered the problem of passing through a door. We will now consider what may happen *in* the room behind the door.
Suppose, one person enters the room to smoke a cigar, while another person enters the room to avoid cigar smokers. Clearly, these two people have conflicting interests and should not be in the room at the same time. Preferably the second person should also precede the first one, for the first one may well contaminate the room with his cigar smoke for a period of time which ex-

ceeds the duration of his presence in the room. There are thus both exclusion and ordering aspects to this problem.
To avoid conflicts inside the room we must give the referee or doorkeeper some assistance. The referee (doorkeeper) only sees to it that people who want to enter the room will eventually succeed without bumping into each other. Now we need someone who can foresee conflicts in the room and avoid them by preventing the entrance of persons to the room when other persons with conflicting interests are still inside. We call such a person a *guard*.
To perform his task the guard must have access to specific information:

(1) the guard must know what activities may be performed in the room, and
(2) what combinations or which sequences of activities may lead to a conflict.

We call this the guard's *semi-permanent knowledge*.
The guard must also know in which 'state' the room is at any instant:

(3) the guard must know which activities are presently being performed in the room, and which activities have been performed (in so far as this is relevant), and
(4) what the people trying to enter the room want to do there.

We call this the guard's *transient knowledge*.
Any person who wants to enter the room must accept and obey the following rules:

(I) No person may stay in the room forever, to prevent others (with conflicting interests) from waiting forever.
(II) Being in the room, one may only perform those activities that were reported to the guard before entering. For new activities the room must first be left and then re-entered via the referee (doorkeeper) and the guard.

When someone desires to enter the room he makes a *request* to the guard (after passing the doorkeeper) stating precisely what activities he wants to perform in the room. The guard considers the request, by inspecting his semi-permanent and transient knowledge. Depending on this knowledge the guard will either acknowledge the request and allow the person to proceed into the room, or he will ask him to enter a waiting room. When someone leaves the guarded room the guard reconsiders all pending requests. This implies that every person leaving a room must explicitly report this to the guard.

Remarks:

(1) The coordination rules enforced by the guard, as described above, can be formalized as a partial ordering on the activities that people want to perform inside the guarded room. With a still more sophisticated guarding strategy, the partial ordering is no longer static but *dynamic*, that is: the partial ordering can be altered by the people served by the guard. To realize this we must give the guard access to information which is also accessible to the people being served.
(2) The working of the guard corresponds in process coordination problems to the working of an exclusive procedure. Access to the exclusive procedure is acquired via special coordination primitives (locks, semaphores, monitors, see chapter 1: **On multiprocessing**) which in turn symbolize the working of the referee or doorkeeper.

The referee passes a request to the guard as soon as the guard is idle, i.e. has finished with his consideration of previous requests. The referee there-

fore has a *crowd* of pending requests, comparable to the guard's waiting room. The referee must evidently know when the guard is idle or busy. The guard must therefore notify the referee whenever it changes its state. In order to keep his transient knowledge up-to-date the guard must not only know which people have entered the room, but also which people have left it. The people must therefore leave the room via the guard, and (as the guard can only perform one function at a time) with the aid of the referee. It is wise to give a leaver a priority over a requester in the refereeing procedure, as the former may well remove a delay-condition for the latter.

A slightly different strategy for the working of the guard is possible. A *lazy guard* will only reconsider a request when it is explicitly repeated. It is then up to the persons trying to enter the room, to repeat their requests until they succeed in acquiring the permission of the lazy guard. With the more active guard outlined above the people deposit their request once, enter the guard's waiting room and fall asleep in the knowledge (or confidence) that the guard will awake them as soon as they can enter the room safely. In fact, a lazy guard may well have to work harder than an active guard, as the lazy guard may be overloaded with repetitive requests.

Guarding is performed in principle for one person's request at a time. (There are ways to relax this restriction though, for non-conflicting requests.)
In figure 4.1 the cooperation between guard and referee (doorkeeper) is illustrated.
We distinguish between:

- The doorkeeper's box, plus waiting 'crowd', which we call the ENTRANCE.
- The guard's room plus waiting room, which we call the RECEPTION.
- The guarded room itself, which we call the CLUB.

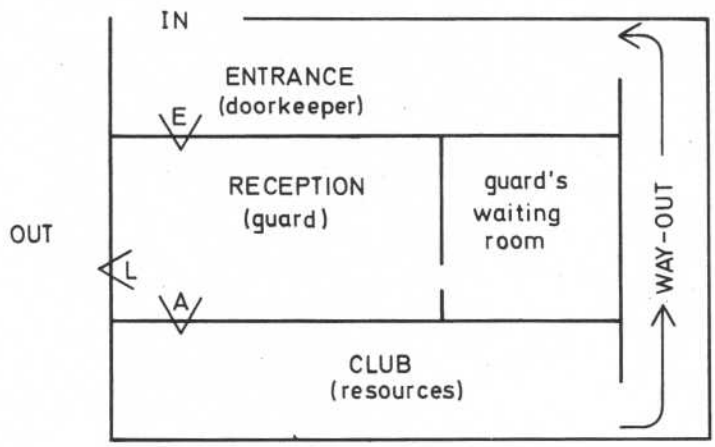


Figure 4.1.
The Analogon of the Guarded Room

- Three doors: from ENTRANCE to RECEPTION (E);
  from RECEPTION to CLUB (A);
  from RECEPTION to OUT (L).
- And a 'conflict-free' path from the CLUB back to the ENTRANCE (for those who leave the guarded room and have to report back to the guard, before leaving via (E) and (L). The door E is controlled by the doorkeeper; doors A and L are controlled by the guard.

Remarks:

(3) The 'lazy guard' implements the busy-waiting strategy discussed in chapter 1; the active guard implements the sleep-wait scheduling strategy.

(4) To perform the tasks of refereeing and guarding, concurrent processes must at least have means to communicate. This implies that there must be some form of exclusion which enables an undisturbed communication, for instance: indivisible read an write operations (see also chapter 1, section 1.2.3: **Arbitration and scheduling**).

(5) We may extend the analogon of the guarded room still further to an analogon of a 'guarded house'. Behind the first door there may then be a hall with a second and a third door and so on. Each door needs refereeing and guarding. When the doors are not too frequently used, all doors may however be controlled by just one referee (doorkeeper) and one guard. Note that if there are more doors, the selections performed by the guard(s) can be ordered, such that each new door symbolizes a further refinement of the selection. Depending on their usage and on the related conflicts one may choose a divisioning of the resources over the rooms in the guarded house. It is plausible that conflicting activities are, in so far as possible, placed in different rooms, so that delays can be minimized.

4.3. NEW CONCEPTUAL MODEL FOR THE STUDY OF PROCESS COORDINATION

4.3.1. Introduction

In this section we want to investigate at what level of abstraction one can best define and enforce the coordination rules in a multiprocessing system. To this purpose we will first try to get a better understanding of the meaning and place of the concept of an abstract *process*, a discussion which was started in the introduction to chapter 1 (**On multiprocessing**)[1].
We have noted that a process should not be interpreted as the mere abstraction of "a processing unit running a program". Such a strong connection between the concepts of a program and a process may be plausible in the sequential programming realm, it is however unnecessarily and undesirably strict in parallel programming.
Horning & Randell '73 gave the following definition of the term 'process', which is independent of the interpretation of the terms 'processor' and 'program'. (Compare this to the definition given by **Denning '71 (pg. 194)**, also quoted in **Müller '76 (pg. 47)**):

> "A process is a triple (S, f, s) where S is a state space, f is an action function **[a mapping from states into actions]** in that state space, and s is the subset of S which defines the initial states of the process."
> **(Horning & Randell '73, pg. 9).**

It is intentionally left unspecified how a formal program relates to 'action function' f. To illustrate this, we can state in the phrasing of **Wirth '69 (pg. 490)** (see chapter 1, section 1.1.1):

> There is a difference between processes and programs. Many programs can be executed in the same process during consecutive periods of time, and vice versa: a single program can be executed by many processes in cooperation (such that each process performs a part of the program).

Specifically a parallel program may relate to a whole set of potentially interacting concurrent processes. Different processes can refer to the same (or identical) program parts.
Consider a multiprocessing system which performs the following functions: It reads a complete record (containing text) from an input-file, edits the text, and writes the edited text in an empty record from an output-file. Assume, there are three concurrent processes: one for reading, one for editing, and one for writing in the output-file. Now, between the lines we have already chosen for a specific interpretation of the processes in this system. We have taken the system's point of view, in which the tasks of the system are logically divided in three parts; each part corresponding to one process. We can also take the user's point of view and distinguish between N processes (if N is the number of records to be read-edited-rewritten), where each process is related to the processing of precisely one of the N records.
This example illustrates that the relation between programs and processes is indeed a rather loose one. (The example is taken from **Brinch Hansen '73.**)

The place of the coordination rules
We will now investigate whether the concepts outlined earlier are sufficient for a clear description of coordination relations, at the appropriate level of abstraction. As argued in chapter 2 (**On correctness analysis**), the coordination relations should not strictly be defined on the level of the individual operations in a program, as one would be forced to do when using lock primitives or

— — — — —

(1) The reader may find it helpful to reread that discussion briefly, before proceeding.

semaphores (see chapter 1, **On multiprocessing**).
The level of abstraction is in general too low for transparent descriptions. On the other hand, the coordination relations should not be defined strictly on the level of the system of processes as an entity, either, as one would be forced to do with (for example) path expressions or in the actor model of computation (see chapter 2, **On correctness analysis**).
For the formalization of coordination relations we need a concept that can be defined on any level of abstraction: as detailed as an operation or as comprehensive as a system of processes. The coordination relations are directly aimed at the *protection* of program parts against unintended interactions during their execution. Knowing the loose relation between the concept of a process and the concept of a program, which is inherent in multiprocessing, we must conclude that it would be inappropriate to define coordination relations on the level of the concurrent *processes* as such. We will not attempt to force and redefine the concept of a process to fit the present model. What we are looking for is a 'missing link' between the concept of a process and the concept of a program. We call this 'missing link' the *section*[1].

A section can consist of just one single operation in a program, and it can be defined as large as an entire system of programs. Coordination rules are defined on the level of the sections in the form of *initiation-conditions*. The enforcement of these initiation conditions will effectuate an ordering of the processes in a system, which may be partially indeterministic.

If we define the sections at the operations level, we can interpret the *sequential* program as a specific collection of sections (where each section corresponds to one specific operation in that sequential program), annotated with initiation conditions. In this case the enforcement of the initiation conditions will result in the complete ordering of the executions of the operations related to the sections. Similarly, if we define the sections on the level of complete sequential programs (as entities), we can interpret a parallel program as a collection of sections and a *partial* ordering as expressed in the initiation conditions.
Sequential programming can thus indeed be understood as a mere special case of parallel programming.

The initiation conditions of the sections can be interpreted as *relations* among sections. These relations are specified by the coordination rules. We are already familiar with some basic types of section-relations in the sequential programming realm: concatenation, selection and iteration. In fact, we can recognize these now as special types of more general orderings. There are many other types of orderings, infinitely many even, though only a small subset of these orderings will be relevant to normal (parallel) programming practice. Some examples are:

- the exclusive execution of two or more sections, in any sequence;
- concurrent (non-exclusive) execution of sections, in any sequence or simultaneously;
- the execution of one section may only be initiated when some other section has been executed ('completed') at least N times, where N is larger than 0.

As long as we have no reliable means to distinguish relevant or practical types of orderings from the non-practical or irrelevant ones, we had better find general descriptive tools with which we can represent any possible type

— — — — —

(1) To avoid possible confusions: the sections of programs are only *named*; the sections of this dissertation are also *numbered*. In some cases we will abbreviate the term when used in the second meaning as 'sect.' .

of ordering in unambiguous terms. We are then able to evaluate their usefulness in a later stage. The approach taken here is therefore essentially different from the one taken by (for instance) **Dijkstra '75**, who restricted himself to only 2 standard control-flow-structures with the 'guarded commands' (the 'alternative construct', and the 'repetitive construct'; we return to this in section 4.3.3).

> We have encountered two common types of coordination relations before: exclusion relations and ordering (sequencing) relations. These two types of coordination relations are, in fact, not fundamentally different. Exclusion can be interpreted as a form of 'indeterministic ordering' (A follows B or B follows A), and sequencing can be interpreted as a 'dynamic exclusion relation' (the exclusion on B is released after the execution of A). It would however be no improvement to the understandability of the descriptions if we would attempt to reduce exclusion relations to ordering relations or vice versa.

To be able to describe coordination relations in the appropriate manner we introduce some additional terminology. We will define sections, atomic sections, structured sections, gate variables, communal variables, formal parameters, coordination constants, and coordination relations (in terms of initiation conditions) on varying levels of complexity.
We start by taking a closer look at 'sections'.

Sections
A section is defined as a program or program-part of which the legality of execution depends on the execution of certain other program(part)s. This 'legality' can be derived from coordination rules and/or system invariants, which can be thought to be part of the coordination *problem* specification. A section may consist of just one operation, or it can be as large as a system of parallel programs.
Example:

> In a sequential program, the operations are to be executed in a specific order, as specified in the program text. The execution of each single operation depends on the progress of the computation. In this case, each single operation is bound via sequencing rules to its environment.
> On a higher level of abstraction there may be a program part ('critical' section) which may not be executed simultaneously with a certain other program part (section). On a still higher level it is possible that the execution of an entire program depends on certain initiation conditions (like: 'data-sets available').

A well-structured system of parallel programs can then be specified as a collection of hierarchically nested sections, plus a well-defined set of coordination relations (initiation conditions). If no coordination relation is specified between two sections, their execution is considered independent.
Process relations are understood in terms of relations among sections. This approach implies another change in our terminology. For example: we no longer say 'process 1 excludes process 2', but 'the execution of section a (by *any* process, including process 1) excludes the execution of section b (by any process, including process 2).
The coordination relations do not discriminate between processes as such, but between sections of which the initiation can be delayed.
Thus the *actions* are being protected, and not the (anonymous) *actors*.
Sections are considered to be independent of the processes executing them. This implies that sections can be executed by several processes simultaneously (if the coordination rules permit this). Observe that sections can be part of cyclical processes, which implies that one process can repeatedly execute a single section.

Atomic sections
Sections which are defined on the lowest level of a nesting hierarchy are called 'atomic sections'. Atomic sections contain no substructures of sections on still lower levels of abstraction.

Each section can be considered as a computational entity with respect to the coordination requirements. We formulate the following *axioms* for sections and atomic sections:

A(1) A section is only initiated if its initiation condition is true.
A(2) The evaluation of an initiation condition and the initiation of the corresponding section is one indivisible act, mutually excluded from all initiations of other sections in the same system.
A(3) The termination of a section as such can not be blocked.
A(4) For each single initiation of an atomic section, this atomic section will terminate precisely once, within a finite and nonzero time.

Remarks:
(1) It is possible to relax the restriction stated in A(2), such that the initiation conditions of sections which belong to the same *conflict-set* may not be evaluated simultaneously in the above sense. We will return to this in section 4.3.3.3, remark (2).
(2) Axiom A(4) can be extended to a larger set of sections: the 'structured sections'. We will elaborate this extension in section 4.6.
(3) The term 'section' was also used by **Ashcroft & Manna '70**, though in a more restrictive sense. Still, their method of simplifying coordination structures in an analysis applies readily to our definitions. The method is to couple those sections that are independent of one another (with respect to the coordination requirements) and to include them in more restrictive structures. Ashcroft & Manna distinguished between the OR termination and the AND termination of sections. Unless otherwise stated, we will refer to the AND termination variant (that is: the termination of a set of sections is defined as the termination of *all* its members). In the appendix we will elaborate also the OR termination variants.
(4) Refering to the analogon of the guarded room, discussed in section 4.2 (**Coordination principles**), we note that:

- A person corresponds to a process; a section corresponds to the (composite) activity the person wants to perform in the room.
- The initiation conditions are evaluated by the guard.
- The presence of the referee (doorkeeper) is implicitly assumed in axiom A(2).
- Axiom A(3) states that no person can be placed in the waiting room of the guard when leaving the guarded room (only when entering).
- Axiom A(4) formalizes one of the rules of the guard (rule I, pg. 174).

4.3.2. Coordination rules

4.3.2.1. *Gate variables*
We introduce two vectors, named I and T, respectively. The number of vector-elements in both I and T is equal to N, the number of sections in the system. The contents (value) of the vector elements is a discrete, non-decreasing function of time.
With $I_t(i)$, where $1 \leq i \leq N$, we indicate the *total number of initiations of section i, since system initialization, at time t.*
With $T_t(i)$, where $1 \leq i \leq N$, we indicate the *total number of terminations of*

section i, since system initialization, at time t.
$I(i)$ and $T(i)$ are called the *gate variables*[1] of section i. The gate variables specify the number of passages through respectively the conceptual 'entry gate' and the conceptual 'exit gate' of a section.
The state of section i at time t can be described by the tuple $(I_t(i), T_t(i))$. (In most cases we will specify states independent of t, though.)

The following axiom can be stated for gate variables:

A(5) Directly after system initialization (at time t = 0) we have:

$(\forall\ i)\ (1 \leqslant i \leqslant N \rightarrow I_o(i) = T_o(i) = 0)$, and for all $t > 0$ we have:
$(\forall\ i)\ (1 \leqslant i \leqslant N \rightarrow I_t(i) \geqslant T_t(i))$.

The state of the system of sections (at time t) can be specified in a set S_t of all section tuples:

$$S_t = \{(I_t(i), T_t(i)) : 1 \leqslant i \leqslant N \cap t \geqslant 0\}.$$

Each initiation and each termination of a section causes a state transition, which is modelled as an increase of one tuple-element of one section-tuple in S_t.
Call **T** the set of all possible state transitions, and let **S** be the set of all possible states. Clearly, **T** is a mapping of a (subset of) **S** into **S**.
If S^1 and S^2 are two states from **S**, the transition $S^1 \rightarrow S^2$ can be indicated as a *transition tuple* (S^1, S^2). Now, (S^1, S^2) is element of **T** only if S^1 and S^2 differ in precisely one element of one section tuple. If we indicate the old state of that tuple by (I^1, T^1) and the new state by (I^2, T^2), then either $I^2 = I^1 + 1$ and $T^2 = T^1$, or $I^2 = I^1$ and $T^2 = T^1 + 1$.
Symbolically:

$$(S^1, S^2) \in \mathbf{T} \text{ iff } S^1, S^2 \in \mathbf{S} \cap \{S^2\} = \{S^1\} + 1).$$

Not all state transitions listed in **T** are legal. For instance, a section may only be terminated when it has been initiated first. This can be inferred from axioms A(4) and A(5). Together with the coordination requirements these restrictions define a subset of **T** which contains all *legal* state transitions.

4.3.2.2. *Communal variables, formal parameters, and coordination constants*
Coordination relations will be expressed with the aid of the following components:

- gate variables,
- coordination constants,
- communal variables,
- formal (also called 'section-' or 'index-') parameters,
- algebraic operations, and
- logical- and relational connectives.

We will find it helpful to use section-identifications with an index-field. For instance, we may define a section named WRITE, with an index which specifies the record in which writing actions are being performed: WRITE(record), in which 'record' is a formal/section/index parameter.
Consider the following rule:

— — — — —
(1) The term 'gate-variable' is borrowed from **Robinson & Holt '73.**

$$((I_{WRITE(record)} - T_{WRITE(record)} \leq \text{constant}) \cap (COMM = TRUE)) \quad [1)]$$

- WRITE is the name of a section;
- record is a formal parameter;
- $I_{WRITE(record)}$ and $T_{WRITE(record)}$ are gate variables;
- constant is a coordination constant, which must have been assigned a value upon system initialization;
- COMM is a communal variable, in this case a boolean.

A communal variable is a variable which can be accessed (and changed) via indivisible operations within section bodies. During the evaluation of an initiation condition, the momentary value of the communal variable(s) is (are) read (by the abstract 'guard'). For each new evaluation, the value of the communal variable(s) must be determined anew.
Note that the communal variables, thus defined, can be used to model all types of dynamic coordination relations. The communal variables extend the modelling power of the section model beyond mere partial orderings (cf. chapter 2, pg.100 and 109).
The formal parameters in coordination rules can be compared to the formal parameters of an ALGOL procedure.
Whenever a process requests the initiation of a section it should specify the actual values of the formal parameter(s) (If present). For obvious reasons we must require that section-parameters will not be changed in section-bodies.
Each section-name with formal parameter corresponds to a set of gate-variables; each actual value of the formal parameter specifies a unique pair of gate-variables.

We will now discuss some basic types of coordination rules, on varying levels of complexity. We begin with 'simple sequencing rules' and 'simple exclusion rules'.

4.3.2.3. *Simple rules*
Simple sequence rules
Simple sequence rules specify and formalize the order in which sections are to be executed. The simple sequence rules are all of the following type:

$$(T(j) - I(i) \Xi n_j),$$

where $j \neq i$, and n_j is a coordination constant and in which Ξ must be one of the relational connectives: $>$ or $\geqslant$.

Example:
Assume $n_j = 1$ and $\Xi = \geqslant$. Directly after system initialization $T(j) = I(i)$ (axiom A(5)). If the rule is implemented as an initiation condition for section i, the effect will be that section i can only be initiated if section j has been executed at least once before. In the initiation condition of section i we can thus include a sequence clause for each section in the system on whose progress the initiation of i depends.

Note that the sequencing symbol ";" from the sequential programming languages specifies a simple sequence rule on a low level of abstraction. In the producer/consumer problem (see chapter 1, section 1.2.1) we can define a similar sequence rule, on a higher level of abstraction.

- - - - -

(1) In this case it is simpler to specify the section-names as subscripts of the gate-variables. We will do so whenever this can cause no confusion.

Simple exclusion rules
Simple exclusion rules specify and formalize (potential) conflicts between sections. Those sections may not be executed concurrently. These rules are all of the following type:

$$(I(j) - T(j) \; \Xi \; n_j),$$

where $n_j \geqslant 0$. Ξ must be one of the following connectives: $<$ or $\leqslant$.

Example:

Assume $n_j = 0$ and $\Xi = \leqslant$. Directly after system initialization $I(j) = T(j)$. If the rule is implemented as an intitiation condition for a section $i \neq j$, the effect will be that section i cannot be initiated while section j is active. The same rule with $n_j = R$ would imply that 'at most' R processes may be executing section j at a time, if the rule is implemented as an initiation condition for section j itself. (R is a coordination constant.)

Rationale
The reasons for restricting the values of n_j and Ξ in the simple rules to those specified are the following.
We have stated in axiom A(3) that the termination of a section cannot be blocked. The coordination rules must be enforced via the initiation conditions, and thus we may require that only section-initiations can make (other) initiation conditions false. If also terminations could make such conditions false, we would be confronted with serious problems in the analysis of the correctness of coordination schemes. In the simple rules defined here, the termination of a section can only make an initiation condition true and never false.
(We return to this point in section 4.5.: **Correctness analysis of coordination schemes.**)

4.3.2.4. *More complex rules*
The simple coordination rules (abbreviated as S-rules) consist of merely 2 gate-variables and 1 coordination constant per clause. In general an initiation condition will be composed of several coordination clauses, of varying levels of complexity. A first extension is to compose rules with other combinations of gate-variables and coordination constants. We call such rules 'median rules' (abbreviated as M-rules). Still, more complex rules, containing also communal variables, will be called 'complex rules' (abbreviated as C-rules). For the M-rules and C-rules it is not always possible to distinguish between exclusion effects and ordering effects. We give explicit rules for the composition of coordination rules of the M- and C-type, in a later section (see sect. 4.5.).

4.3.2.5. *Introductory example*
As an introductory example of the use of general coordination rules to devise coordination schemes, we discuss the third version of the readers' and writers' problem (see chapter 1, pg. 43). We will describe the solution which was illustrated with an 'extended exclusion graph' in figure 30 of chapter 2 (**On correctness analysis**).
The readers' program consists of two nested sections: WR and AR, and similarly the writers' program consists of two nested sections: WW and AW.
First we have the ordering property that AR cannot initiate before WR has initiated, and similarly for AW and WW. This yields an initiation clause for sections AR and AW. We indicate the clauses with the symbol cic, subscripted with the section-name, and superscripted with a number indicating the number of the clause. If the clause is a clear exclusion clause we notate it as: eic, if the clause is a clear ordering clause we notate it as: sic. We found:

$$(1)\ sic^1_{AR} = ((I_{WR} - T_{WR}) - (I_{AR} - T_{AR}) > 0)$$

or equivalently (as we may take T_{WR} equal to T_{AR}):

$$sic^1_{AR} \hat{=} (I_{WR} - I_{AR} > 0)$$

$$(2)\ sic^1_{AW} = ((I_{WW} - T_{WW}) - (I_{AW} - T_{AW}) > 0) \hat{=} (I_{WW} - I_{AW} > 0).$$

Both clauses are M-rules, clearly.
Reading and writing must be mutually exclusive. This yields the clauses:

$$(3)\ eic^1_{AR} = (I_{AW} - T_{AW} = 0)$$

$$(4)\ eic^1_{AW} = (I_{AR} - T_{AR} = 0).$$

Both clauses are S-rules.
Writing must exclude writing, so:

$$(5)\ eic^2_{AW} = (I_{AW} - T_{AW} = 0).$$

This is also an S-rule.
To avoid starvation of both readers and writers we can make the following tentative rules (waiting writers exclude the initiation of active readers; waiting readers exclude the initiation of active writers).

$$(6)\ eic^2_{AR} = (I_{WW} - I_{AW} = 0)$$

$$(7)\ eic^3_{AW} = (I_{WR} - I_{AR} = 0).$$

But, clearly these two rules cannot be enforced in this form, as they may lead to a deadlock when both readers *and* writers are waiting.
To avoid the deadlock we can think of an additional C-rule with a boolean communal variable PREF (preference). At the end of section AR PREF can be set to the value FALSE, and at the end of section AW PREF can be set to TRUE. The setting and resetting of PREF must clearly be mutually excluded from all evaluations of initiation conditions in which PREF occurs (we return to this in section 4.5.). We amend rules (6) and (7) as follows:

$$(6')\ eic^2_{AR} = (I_{WW} - I_{AW} = 0)\ \text{OR}\ (\text{PREF} = \text{TRUE})$$

$$(7')\ eic^3_{AW} = (I_{WR} - I_{AR} = 0)\ \text{OR}\ (\text{PREF} = \text{FALSE}).$$

An equivalent solution can also be constructed with only M-rules, as follows:

$$(6'')\ eic^2_{AR} = (I_{WW} - I_{AW} = 0)\ \text{OR}\ (T_{AW} > T_{AR})$$

$$(7'')\ eic^3_{AW} = (I_{WR} - I_{AR} = 0)\ \text{OR}\ (T_{AW} \leqslant T_{AR}).$$

To avoid 'out of bounds' the gate-variables T_{AW} and T_{AR} in the last clause of (6'') and (7'') may be taken MODULO some (large) number N.
Still another possibility is to use the solution described in chapter 2, section 2.3.3. **Path expressions**.

The complete initiation condition for section AR is now:

$$ic_{AR} = sic^1_{AR} \cap eic^1_{AR} \cap eic^2_{AR},$$

and similarly for AW we have found:

$$ic_{AW} = sic^1_{AW} \cap eic^1_{AW} \cap eic^2_{AW} \cap eic^3_{AW}.$$

4.3.2.6. *Classifications*

In some cases a classification of coordination relations may help to reveal the structure in a large set of coordination rules. On the lowest level of such a classification one may place all *reflexive* types of relations. A simple example of a reflexive relation for atomic sections is the S-rule for sequencing (see section 4.3.2.3). The initiation of the section i with ic_i = (T(j) - I(i) = 1) can make ic_i false itself (boomerang effect).
One level higher one can form *classes* of sections with identical or equivalent coordination relations, both with sections in the class considered, and with sections outside that class. We can thus make a classification of sections.

Example:

If one class contains N sections, and another class contains M sections, then all (N.(N - 1)/2) internal class relations in the first class, and all (M.(M - 1)/2) internal class relations in the second class, must be equivalent, and similarly for all N.M coordination relations between the sections of the two classes.

In much the same way one can form main-classes and so on. As a further example of a classification one may consider the first version of the readers/writers' problem (reader priority, see chapter 1). We may define two sections AR and AW, each with index-parameter: AR(record) and AW(record). The coordination relations can then be specified for single records in memory. The exclusion rule for writer-sections (AW(record); compare to rule (5) in section 4.3.2.5) is clearly a reflexive relation. In a first step we may therefore combine all sections AW(record) for all different values of formal parameter 'record' in one class. One level higher we can combine also all reader sections in one class. The two classes, thus defined, have 2 simple exclusion relations between them.

We will not elaborate the classification procedures further here, nor the precise interpretation of the term 'equivalence' in this context. In most cases the coordination schemes written in the language of the section model are readily understandable and require no explicit classifications.

4.3.3. Formalization of the section model

4.3.3.1. *Introduction*

At this point we still abstract from the way in which the coordination rules will be enforced in an implementation of the initiation conditions (see section 4.8). In this section (4.3.3) we will consider how the section model can be formalized without appealing to specific notions about the way in which the initiation conditions can be implemented with lower level synchronizing primitives. We will formalize the model on the level at which it was defined.

We assume the existence of an abstract machine which will enforce the obeyance of the rules. We call this machine *the guard*.
Each section must be declared to the guard, specifying its name and initiation-conditions. The working of the guard is non-interruptable, specifically it cannot be interrupted by the manipulation of communal variables in section bodies. The guard must know the exact state (S_t) of the system at all times, as expressed in the values of gate-variables, communal variables etc.

For simplicity we will assume that *simple sequence rules* can be specified implicitly with the use of the standard sequencing symbol ";". Similarly we will assume that all standard control-flow structures, like *conditional selection*, *iteration*, and *parallel paths* (i.e. COBEGIN/COEND structures), can be specified implicitly with the usual control-flow symbols.

(See also sect. 4.6)[1].
Each section-name must be unique in the system. Each section must have a *head* and a *tail*. In the head we must specify the section-name and the rules (the initiation condition); in each tail we must specify the section-name. We choose for the following notation:

```
SECTION(name);
     RULES: (initiation-condition);
     begin
          (section-body)
     end
ENDSECTION(name)
```

For simplicity we may abbreviate expressions of the type (I_{name} - T_{name}) with the single term 'name'. The solution to the third readers/writers' problem discussed in section 4.3.2.5 can then be written transparently as follows:

```
SECTION(WR);
     SECTION(AR);
          RULES: (AW = 0) AND
                 (WW - AW = 0 OR PREF = TRUE);
          begin
               read;
               PREF := FALSE
          end
     ENDSECTION(AR)
ENDSECTION(WR) |


SECTION(WW);
     SECTION(AW);
          RULES: (AR = 0) AND (AW = 0) AND
                 (WR - AR = 0 OR PREF = FALSE);
          begin
               write;
               PREF := TRUE
          end
     ENDSECTION(AW)
ENDSECTION(WW).
```

Observe that the sequence clauses could be deleted as they are specified implicitly in the nesting order of the sections. The initiation-conditions of sections WR and WW are both TRUE and are deleted. The symbol | is an 'indeterministic control-flow symbol'. The symbol indicates that the code separated by it can be executed in any order or concurrently.

4.3.3.2. *Syntax*
A coordination scheme can be declared in a concurrent program in a monitor-like structure (see chapter 1). The declarations of the sections should be preceded by the declaration and initialization of all communal variables. We use the key-word "braces": SECTIONS and ENDSECTIONS. In BNF[2] notation we can now describe the desired syntax as follows:

— — — — —

(1) Note that the selection- and iteration-predicates are then implicit communal variables.

(2) Backus-Naur Form.

<coordination scheme> ::= SECTIONS {VAR <communal variable> : COMMUNAL <type>;
<communal variable> := <initial value>;} <section> {|<section>} ENDSECTIONS

The braces indicate: 'followed by zero or more instances of the enclosed'.

<section> ::= SECTION(<name>); RULES: <initiation condition>;
section body
ENDSECTION(<name>)

The section body may be empty; it may contain a single statement or complete (concurrent) programs. The programs may be composed of only the four following basic control-flow structures: conditional selection, concatenation, iteration and parallel paths. As statements in those programs one may again use sections or entire coordination schemes as functional entities.
The complete expansion of the syntax rules for coordination schemes is tedious but fairly straightforward. We delete these expansions here for brevity. To make composite manipulations of communal variables in section bodies indivisible, one uses the key-word EFFECT, as follows:

EFFECT(.... manipulation of communal variables ....);

The code placed between the braces of an EFFECT statement is to be protected from interference with guard actions. This can be implemented with simple locking tools.
Note that the actual 'monitoring' functions are not performed in the SECTIONS declaration, but in the guard procedures, to be considered below.

4.3.3.3. *Guard procedures*
It is the task of the guard (procedures) to update the state S_t for each state transition, and to enforce the obeyance of the coordination rules. To enforce the rules the guard may make use of the reduced set **T**, discussed earlier (see section 4.3.2.1). It would however be an unnecessary divertion to translate the initiation conditions to a reduction of the full set **T** first, and then to translate the reduced set **T** to new initiation conditions. The guard can use the initiation conditions directly to determine the legality of a transition (i.e. initiation). The guard procedures are invoked on two occasions: in section-headings and in section-tails.
The execution of a section may be requested by any process in the coordinated system. Each new initiation of a section will cause the creation of a new set of variables which are local to the section body (just like with re-entrant ALGOL procedures). One may consider the extension of the model with explicit parameter passing mechanisms between sections and processes. We will however not study such extensions here.

Directly after the declaration of the coordination scheme, the state space of the guard can be constructed. For each single section the guard reserves two vector-elements, both initialized to zero (the gate variables). The set of communal variables, properly initialized, is added.
After a call on the guard procedures in a section-heading, the initiation condition of that section is evaluated (in exclusive mode). If the condition is TRUE, the calling process is allowed to proceed. If the condition is FALSE the calling process is suspended (put 'asleep'), the initiation condition is amended to an evaluation list ('the waiting room'), and the section-name and the identity of the calling process are stored in a second list.
If the calling process is allowed to proceed (see above) the corresponding gate variable I(section-name) is increased by 1, before the exclusion on the action is released: evaluation and initiation is thus one indivisible action in the guard procedures.

After a call on the guard procedures in a section tail, the corresponding gate variable T(section-name) is increased by one and the process is allowed to continue.
We need one more procedure in which all initiation-conditions placed on the evaluation list are evaluated. We call this procedure (the kernel of the coordination structure) the *bodyguard*.
The bodyguard can be described with a type of 'repetitive construct', as defined by **Dijkstra '75**. The lay-out of the bodyguard is then as follows:

```
BODYGUARD;
begin doo condition(section-name) →
              I(section-name) += 1 ; delete condition(section-name) from
              evaluation-list; restart calling process end;
    -         .....
     ood
end
```

The working of the bodyguard is as follows. The conditions in the evaluation-list are considered one by one until a condition is found that has the value TRUE. In that case the scanning of the list is temporarily suspended, and the code after the arrow in the list is executed. The exclusion on the guard actions is set directly before an evaluation starts and reset directly after the evaluation if the result was FALSE. If the result was TRUE the exclusion is only reset after the execution of the corresponding code placed after the arrow.
The main difference with the repetitive construct defined by Dijkstra is that the evaluation list is scanned in one specific order (from top to bottom). For this reason we have used the key-words **doo** and **ood**, instead of **do** and **od** (**doo** is short for '**do-ordered**'). Observe that this ordering guarantees the obeyance of the fairness and symmetry requirements discussed in sect. 4.5.
The bodyguard is initiated after each guard invocation in section-tails, and directly after the execution of an EFFECT statement in section bodies. In most cases only these actions can make initiation conditions TRUE. The bodyguard terminates only when all initiation conditions are FALSE.

Remarks:

(1) The guard could restrict itself to the re-evaluation of only a subset of the conditions in the evaluation list. In general, one specific action (e.g. the termination of a section) can only affect the truth of a small number of conditions, cf. **Sintzoff & Van Lamsweerde '74**.

(2) The bodyguard, as described above, evaluates the initiation conditions strictly one by one. This restriction can be loosened somewhat if we consider under which conditions two initiation conditions may be evaluated concurrently. For instance, if the initiation of section A can make the initiation condition of section B FALSE, then A can be said to be *in conflict* with B. Formally, we can say that the pair (A,B) is a member of a conflict set Ψ. If neither the pair (A,B) nor the pair (B,A) is in set Ψ, then ic_A and ic_B may be evaluated concurrently. In all other cases the FIFO order of the evaluation list in the bodyguard must prevail.

4.3.3.4. *Example: disk-head scheduling*
As a further example of a formalized notation of a coordination scheme, with the aid of the model developed here, we discuss a disk-head scheduling problem. **Hoare '74 (pg. 555/556)** described the problem as follows:

> "On a moving head disk, the time taken to move the heads increases monotonically with the distance traveled. If several programs wish

to move the heads, the average waiting time can be reduced by selecting, first, the program which wishes to move them the shortest distance. But unfortunately this policy is subject to an instability, since a program wishing to access a cylinder at one edge of the disk can be indefinitely overtaken by programs operating at the other edge or the middle.
A solution to this is to minimize the frequency of change of direction of movement of the heads. At any time, the heads are kept moving in a given direction, and they service the program requesting the nearest cylinder in that direction. If there is no such request, the direction changes, and the heads make another sweep across the surface of the disk."
(Hoare '74, pg. 555).

Hoare gave a solution with monitor-procedures. For comparison we will reproduce Hoare's solution first.
There are two monitor-procedures: *request(dest:cylinder);* and *release;* .
where

type *cylinder = 0 .. cylmax;*

The monitor *diskhead* is declared as follows:

```
diskhead:monitor
begin headpos:cylinder;
      direction:(up,down);
      busy:Boolean;
      upsweep,downsweep:condition;
  procedure request(dest:cylinder);
    begin if busy then
      {if headpos < dest OR (headpos = dest AND direction = up)
         then upsweep.wait(dest)
         else downsweep.wait(cylmax-dest)};
       busy := true; headpos := dest
    end request;

  procedure release;
    begin busy := false;
      if direction = up then
         {if upsweep.queue then upsweep.signal
                           else {direction := down; downsweep.signal}}
      else
          if downsweep.queue then downsweep.signal
                             else {direction := up; upsweep.signal}
  end release;

  headpos := 0; direction := up; busy := false

end diskhead;
```

In the language of the section model we can write essentially the same solution as follows. First we give the declarations of the communal variables. Observe that each declared variable is initialized immediately.

```
VAR headpos: COMMUNAL 1 .. cylmax; headpos := 1;
VAR direction: COMMUNAL (up,down); direction := up;
VAR dest: COMMUNAL array 1 .. N of 0 .. cylmax;
    for j from 1 to N do dest(j) := 0;
```

We will not use a Boolean variable *busy*, and no 'condition' queues. In the following description each process can indicate its 'destination' on the disk

in a seperate element of array *dest*. There are assumed to be (no more than) *N* processes. The section *disk-user*, with formal parameter *i*, can now be declared as follows:

```
SECTION(disk-user(i: 1 .. N));
  RULES:
    (∀j)[1 ≤ j ≤ N → disk-user(j) = 0] AND
    {(direction = up AND dest(i) ≥ headpos AND
       (∀j) [ 1 ≤ j ≤ N → (dest(j) < headpos OR dest(j) > dest(i) OR
                           (j > i AND dest(j) = dest(i)))])
    OR
    (direction = down AND dest(i) ≤ headpos AND
       (∀j) [ 1 ≤ j ≤ N → (dest(j) > headpos OR dest(j) < dest(i) OR
                           (j > i AND dest(j) = dest(i)))])};

  EFFECT(headpos := dest(i));

  read-disk;

  EFFECT(dest(i) := 0;
         if (∀j) [1 ≤ j ≤ N AND dest(j) ≠ 0 → dest(j) < headpos]
         then direction := down
         else
          if (∀j)[ 1 ≤ j ≤ N AND dest(j) ≠ 0 → dest(j) > headpos]
          then direction := up);

ENDSECTION(disk-user)
```

We have taken the liberty to notate the conditions in a Boolean notation, as clearly this lay-out is intended for analysis purposes, and not for an execution on a computer.
We have used the same simplified notation as in section 4.3.3.1 (i.e. we use the name of a section to represent expressions of the type (I_{name} - T_{name})).

Observe that in the section-model notation all coordination relations are made explicit in the RULES part, while all updatings (other than the usual updatings of gate variables) are made explicit in EFFECT statements.
It is no longer relevant whether the disk-user should wait for an *upsweep* or a *downsweep* of the disk-head.

4.4. GRAPHICAL REPRESENTATION OF COORDINATION SCHEMES

We introduce multi-logic graphs for the graphical representation of coordination schemes. Each vertex in the graphs represents a specific section; the initiation conditions are represented by (sets of) directed branches from vertex to vertex. The graphs introduced here can be interpreted as an extension of the 'Coordination Graphs', discussed in sect. 2.2.5.

The effects of the enforcement of a coordination structure can transparently be represented in two graphs:

- a *blocking graph*, and
- a *releasing graph*.

Formally, a blocking graph is a tuple (V,A), where V is a set of vertices and A is a set of arcs. Each section is associated with a vertex, each section-relation is associated with a (set of) arc(s). For each section S_1 of which the initiation *can* make the initiation condition of a section S_2 FALSE, one draws an arc in the blocking graph from S_1 to S_2. We allow for the combination of arcs in multi-headed and/or multi-tailed arcs. At the point where two or more arcs are combined in one multi-arc one can indicate the type of evaluation-logic (for the associated conditions) desired (AND or OR logic).

The dual of the blocking graph is the releasing graph. Formally the releasing graph is a tuple (V,A'), where V is the set of vertices and A' is a new set of arcs. For each section S_1 of which the termination *can* make the initiation condition of a section S_2 TRUE, one draws an arc in the releasing graph from S_1 to S_2. We allow again for multi-arcs and multi-logic. If no evaluation logic is specified it is assumed to be AND logic (in both types of graphs).

Remark:

With M- and C-rules it is possible that also initiations of sections can have a releasing effect, and under some conditions (to be discussed in section 4.5) that terminations can have a blocking effect. In those cases one may annotate the blocking and releasing graphs with symbols I and T, to make the distinction. The default value for the annotations is clearly I in blocking graphs, and T in releasing graphs.

The effects of the communal variables remain to be considered. We adopt the rule that for any section which updates a communal variable, we add a tag to the corresponding vertex in the graphs, specifying the effect.
To represent the blocking and releasing effects of communal variables we add a special type of input-arc to the relevant vertices. This input-arc has no 'source-vertex'. Instead we draw a square box as its source, annotated with the desired blocking or enabling condition.

Examples and simplifications
As an example consider the representation of the simple exclusion rule for two sections A and B:

$$ic_A = (I(B) - T(B) = 0).$$

The blocking graph and the releasing graph are given in figure 4.2.
The S-rule is represented by one arc in each graph, both directed from B to A. We will use a simplified graph for the S-rules. The simple exclusion relation is represented in a *general graph* by an arc with a solid top, as illustrated in figure 4.3. In the general case when:

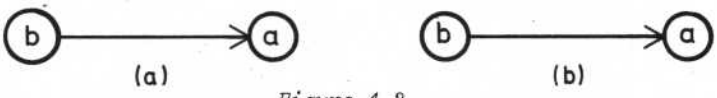


Figure 4.2.
Blocking (a) and Releasing (b) Graph
for Simple Exclusion Rule

Figure 4.3.
General Graph for
Simple Exclusion

Figure 4.4.
General Graph for
Simple Mutual Exclusion

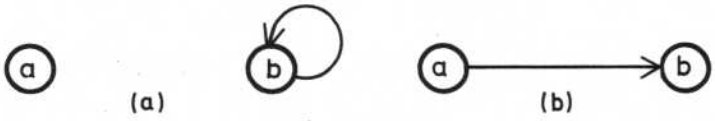


Figure 4.5.
Blocking (a) and Releasing (b) Graph
for Simple Sequence Rule

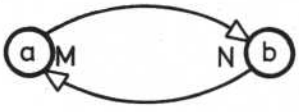
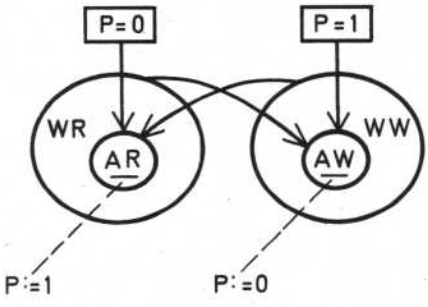


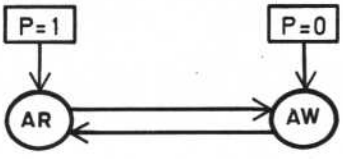
Figure 4.6.
General Graph for
Simple Mutual
Sequence Rules

(a)

Figure 4.7.
Third Readers' and Writers' Problem,
(a) General Graph,
(b) Blocking Graph,
(c) Releasing Graph.



(b)



(c)

$$ic_A = (I(B) - T(B) \leqslant N),$$

with N non-negative, we annotate the exclusion arc with the number N, at the top. The mutual exclusion relation:

$$ic_A = (I(B) - T(B) \leqslant N), \text{ and}$$

$$ic_B = (I(A) - T(A) \leqslant M),$$

with $N \geqslant 0$, $M \geqslant 0$ is represented by the graph in figure 4.4.

The full blocking and releasing graphs for the simple sequence relation:

$$ic_B = (T(A) - I(B) \geqslant 1),$$

are indicated in figure 4.5. We use a simplified representation of this basic rule in a general graph by an arc with a closed non-solid top. In the general case when:

$$ic_B = (T(A) - I(B) \geqslant N),$$

with N non-negative, we annotate the sequence arc with the number N, at the top. A 'mutual sequencing relation', as occurs for instance in the producer/consumer problems, with the general relations:

$$ic_B = (T(A) - I(B) \geqslant N), \text{ and}$$

$$ic_A = (T(B) - I(A) \geqslant M),$$

with $N \geqslant 0$, $M < 0$ is represented in the graph of figure 4.6.

As a last example we consider the graphical representation of the coordination schemes for the readers/writers' problems. It is interesting that the 'general graph' (with all S-rules) is the same for all versions of the readers/writers' problems. This graph is given in figure 4.7.a. We give the additional blocking and releasing graphs for the (most extensive) third version in figures 4.7.b and 4.7.c.
There are many situations where a coordination scheme consists of simple rules extended with simple evaluations of the values of communal variables. The effect of these communal variables is usually complementary in blocking and releasing graphs. In these cases it suffices to indicate one of these effects (e.g. the enabling condition) in the general graph. Examples of this simplified notation are given in figures 4.8 and 4.12.

4.5. CORRECTNESS ANALYSIS OF COORDINATION SCHEMES

We will now consider under what conditions the working of the guard procedures (notably the 'bodyguard') will guarantee fairness and symmetry (as defined in chapter 1, see appendix A). We will formulate some rules for the manipulation of communal variables and for the composition of coordination rules, and finally we will consider in what manner one can analyze the *consistency* of the coordination rules with respect to deadlocks, exclusions, starvations, etc.

Fairness and symmetry
Consider N processes requesting access to N mutually exclusive sections via the guard procedures, at indistinguishable moments.
The guard procedures are indivisible and mutually excluded, according to the definitions given in section 4.3.3.3. The specific way in which the necessary exclusion (on the guard procedures) is realized depends on the implementation of the referee (see section 4.2.1). Let us assume that the refereeing process is implemented such that the fairness and symmetry requirements are obeyed, at least at *that* level. Now, if the refereeing process is indeed fair and symmetrical, the *order* in which the processes' requests are listed in the guard's evaluation-list is fair and symmetrical. The list is searched in this order, and therefore the order in which the process requests are acknowledged is by necessity again fair and symmetrical. Observe that when one process from the group of N processes considered here renews its request for access to a critical section after one cycle, its request is listed *after* all pending requests, at the bottom of the evaluation list.

Composition rules
Quite subtle coordination errors may result if the truth of an initiation condition can be made false without the guard's consent. Note that legal initiations by the guard can be 'surpassed' in this manner. This is possible on two occasions:

(1) via section terminations, and
(2) via the manipulation of communal variables.

To prevent the errors we formulate the following composition (or design) rules:

(1) *The termination of a section may not falsify any initiation condition, unless it can be proved that it is irrelevant whether or not such a falsification surpasses a legal initiation of the corresponding section, OR unless it can be proved that such falsifications cannot surpass legal initiations of the corresponding sections.*

(2) *The effect of the updating of communal variables may not falsify any initiation condition, unless it can be proved that it is irrelevant whether or not such a falsification surpasses a legal initiation of the corresponding section, OR unless it can be proved that such falsifications cannot surpass legal initiations of the corresponding sections.*

The observance of these design rules will guarantee the integrity of the guard procedures, though it does not guarantee the consistency of the coordination rules as such. Note that violations of these rules can be 'repaired' by the addition of simple exclusion rules to the coordination scheme.

Let us, as an example, consider the coordination scheme for the third version of the readers/writers' problem, discussed in a.o. section 4.3.2.5.
The readers can falsify their own initiation condition via communal variable

PREF. We must verify that rule 2 is obeyed. The access to PREF is trivially harmless. Observe that the execution of writers is blocked until all executing readers have completed.

Consistency of coordination rules
We distinguish between three items in the consistency analysis:

(1) Undesirable blockings,
(2) Desirable exclusions, and
(3) Reachability of system states.

In a full analysis one may include still other criteria, but here we will concentrate on these three basic points. It is fairly straightforward to analyze a coordination scheme on these points, with an informal procedure. (An extension to formal methods which can be automized prompts itself. We will return to this point in section 4.9, and in appendix D.)

(1) Undesirable blockings
A section is 'undesirably blocked' if its initiation condition can become FALSE permanently. We need not distinguish between deadlocks and starvation loops here.
As an example of the analysis procedure we consider the third version of the readers/writers' problem, the producer/consumer problem, and the cigarette smokers' problem (all introduced in chapter 1, **On multiprocessing**).

a. *Readers/writers' problem*
We consider whether readers can be blocked indefinitely (section AR).
We have:

$$ic_{AR} = (AW = 0) \text{ AND } (WW - AW = 0 \text{ OR } PREF = TRUE).$$

Directly after system initialization ic_{AR} is true as AW = WW = 0.
The blocking condition is:

$$(AW > 0) \text{ OR } (WW - AW > 0 \text{ AND } PREF = FALSE) = TRUE \text{ permanently.}$$

There are three possibilities: either the first clause is permanently TRUE, or the second (composite) clause is permanently TRUE, or the two clauses are alternatingly (and overlappingly) TRUE.
It is easy to show that $0 \leqslant AW \leqslant 1$ (using ic_{AW}), and as AW is an atomic section AW cannot be permanently larger than zero. In the interval between the leaving of one writer and the access of another AW = 0, and therefore the first clause cannot be true permanently.
The second clause can only be TRUE permanently if writers are blocked as well. (Observe that PREF is set to TRUE in the first execution of AW.) The blocking condition for writers is:

$$(AR > 0) \text{ OR } (AW > 0) \text{ OR } (WR - AR > 0 \text{ AND } PREF = TRUE) = TRUE \text{ permanently.}$$

Readers are assumed to be blocked, as well as writers, so within finite time AR = AW = 0, which reduces the blocking condition for writers to:

$$(WR > 0 \text{ AND } PREF = TRUE) = TRUE \text{ permanently.}$$

Similarly the second clause in the blocking condition for readers reduces to:

$$(WW > 0 \text{ AND } PREF = FALSE) = TRUE \text{ permanently.}$$

Clearly, the two conditions cannot both be true at the same time, as PREF is either TRUE or FALSE at any instant. This proves that the second clause in the blocking condition for readers cannot block the readers indefinitely.
It remains to consider an overlapping block by the two clauses.
As PREF is only set to FALSE within AR, section AW must again be blocked as well. (Observe that the first execution of AW would set PREF to TRUE, and PREF would remain TRUE forever, due to the block on AR.) This would imply that AW = 0 within finite time, permanently, and this contradicts the assumption. The conclusion must be that section AR cannot be blocked indefinitely.
A similar argument can be given for section AW. Sections WW and WR are never inhibited and thus never blocked.

b. *Producer/consumer problem*
We consider whether the producer can be blocked indefinitely. We have:

$$sic_P = (\tilde{T}(C) - I(P) > -B), \text{ and}$$

$$sic_C = (T(P) - I(C) \geq 1),$$

where B is the length of the buffer between producer and consumer.
We further have the exclusion clauses:

$$eic_P = eic_C = (C = 0) \text{ AND } (P = 0)$$

to protect the access to the shared buffer.
The blocking condition for P is:

$$(C > 0) \text{ OR } (T(C) - I(P) \leq -B) = \text{TRUE permanently.}$$

There are again three possibilities. The first clause can be permanently TRUE, the second clause can be permanently TRUE, or the two clauses can alternatingly be TRUE.
Suppose that the first clause is permanently TRUE. It is again easy to show that $0 \leq C \leq 1$, which implies that this would be impossible. Furthermore, the consumer will 'overtake' the producer within B cycles, thus falsifying its own initiation condition which makes C = 0 within finite time.
Suppose the second clause is permanently TRUE. Within finite time the initiation condition of the consumer must become TRUE, and within a finite number of executions of the consumer the second clause in the producer's blocking condition must become FALSE.
It remains to consider an overlapping block. An overlapping block would imply that the consumer executes repeatedly (making C larger than zero). But this would make the second clause in the blocking condition FALSE within B executions as the producer is assumed to be blocked.
The conclusion must be that section P cannot be blocked indefinitely. A similar argument can be given for section C.

c. *Cigarette smokers' problem*
A description of the cigarette smokers' problem in the section model can be given as follows. There is one supplier S (the 'agent') and there are 3 smokers : T (with tobacco), M (with matches), and P (with paper).

$$ic_S = (T = M = P = 0) \text{ AND } (PAPER = MATCHES = TOBACCO = FALSE)$$

$$ic_P = (S = 0) \text{ AND } (MATCHES = TOBACCO = TRUE)$$

$$ic_M = (S = 0) \text{ AND } (PAPER = TOBACCO = TRUE)$$

$$ic_T = (S = 0) \text{ AND } (PAPER = MATCHES = TRUE)$$

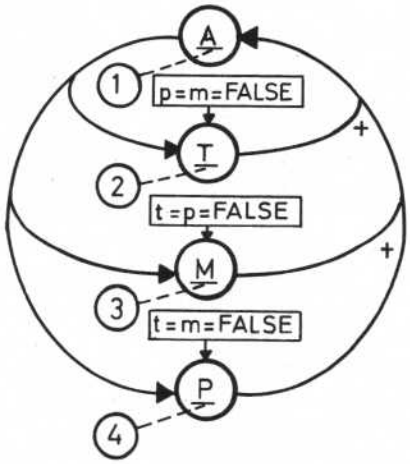


Figure 4.8.
General Graph with Blocking Conditions
for Patil's "Cigarette Smokers' Problem".
EFFECT clauses:

```
(1) (p := m := TRUE) OR (t := p := TRUE) OR
    (t := m := TRUE)
(2) (p := m := FALSE)
(3) (t := p := FALSE)
(4) (t := m := FALSE)
```

There are three communal variables: PAPER, MATCHES, and TOBACCO, all initialized to FALSE. The supplier selects two of the tree ingredients at random, and makes the corresponding communal variables TRUE. The smokers set the communal variables corresponding to the ingredients they have received from the supplier to FALSE in their smoker sections. The general graph for this coordination scheme is given in figure 4.8.
(The graph is surprisingly simple and transparent. Compare it for example to the solutions of **Lauer & Campbell '75**, which are extremely complicated.)

Let us consider whether smoker M can be blocked forever. The blocking condition is:

(S > 0) OR (PAPER OR TOBACCO = FALSE) = TRUE permanently.

Due to the indeterministic behaviour of the supplier the second clause of this condition may well remain FALSE forever, for instance, when smoker P is favored systematically. Still, the three smokers cannot all three together be blocked forever, on these clauses, as this would imply:

(PAPER AND MATCHES = FALSE OR PAPER AND TOBACCO = FALSE OR
MATCHES AND TOBACCO = FALSE) OR (S > 0)

This can only occur if the supplier is blocked forever, and this in turn implies that one of the smokers must be active. The latter however contradicts the assumption that all smokers were blocked.

(2) **Exclusions and reachability**
It is particularly easy to show that desirable exclusions are present in a well-designed coordination scheme. That readers and writers cannot execute concurrently in the third version of the readers/writers' problem follows directly from the initiation conditions of the corresponding sections. It is a trivial task to show that a state in which AR > 0 AND AW > 0 can only be reached by a violation of axiom A(1), or a violation of one of the design rules.
It is a similarly straightforward task to show that any desirable system state (e.g. R simultaneous executions of AR: AR = R) can be reached from the initial state and vice versa. For brevity we do not elaborate this here.

Note that an analysis procedure as described here can only show that the coordination rules are consistent with respect to deadlocks, starvations, exclusions and the like. It cannot show that the coordination rules do adequately model the actual problem to be solved. No matter how rigorous an analysis is, at some initial point a designer of a coordination scheme must make a first formalization of the coordination requirements. In the process of removing the ambiguities from an informal problem statement this designer could well make mistakes. This cannot be avoided, precisely because the original informal problem statement contains the ambiguities.
As argued in chapter 2, we will not discuss the functional correctness aspects for section-bodies. Those aspects belong mainly to the sequential programming realm, and as such they fall beyond the scope of this study. In this chapter we restrict ourselves explicitly to the analysis of coordination properties as such.

4.6. BASIC SEQUENCING STRUCTURES

In section 4.3.3.1 we have stated that concatenation, conditional selection, iteration, and parallel paths are such basic sequencing structures that they can be specified implicitly with the standard control-flow key-words. In this section we consider how these basic structures can be made explicit in coordination rules. We study some properties of these structures, and use these for the definition of a special type of sections, which can be considered as the equivalents of the atomic sections defined earlier: the *structured sections*.
We begin with a description of the explicit coordination rules for the four basic structures mentioned.

(1) Concatenation
A *processing line* is defined here as an ordered set of sections:

$$\{a_1, a_2, \ldots, a_m\},$$

with initial state: $I(a_1) = T(a_1) = I(a_2) = \ldots = T(a_m)$,

and sequence relations:

$$(\forall i)(1 < i \leqslant m) \rightarrow \left(sic_{a_i} = (T(a_{i-1}) - I(a_i) \geqslant 1)\right)$$

sic_{a_1} is unspecified.
ic_{a_1} is called the initiation condition of the processing line.
Section a_1 is called the *initial section* of the line.
Section a_m is called the *terminal section* of the line.

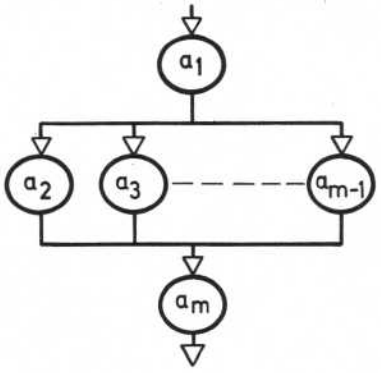


Figure 4.9.
Selection Structure

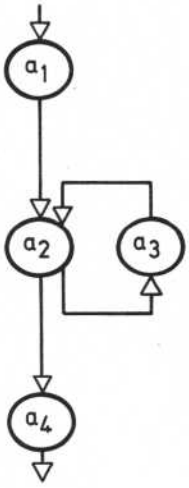


Figure 4.10.
Iteration Structure

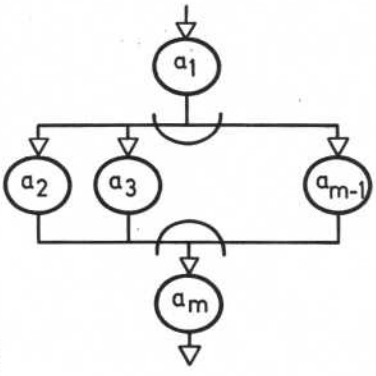


Figure 4.11.
Parallel Path

(2) **Selection**
A *selection structure* is defined as an ordered set of sections:

$$\{a_1, a_2, \ldots, a_m\},$$

with initial state:

$$\left(T(a_1) = \sum_{i=2}^{m-1} I(a_i) = I(a_m)\right) \text{ AND } (\forall\, i)((1 \leqslant i \leqslant m) \rightarrow I(a_i) = T(a_i))$$

and sequence relations:

$$(\forall\, i)\ ((1 < i < m) \rightarrow sic_{a_i} = \left(T(a_1) - \sum_{j=2}^{m-1} I(a_j) \geqslant 1\right) \text{ AND}$$

$$sic_{a_m} = \left(\left(\sum_{j=2}^{m-1} T(a_j)\right) - I(a_m) \geqslant 1\right).$$

The selection criterion is specified in another set of initiation clauses to the sections $a_2, \ldots a_{m-1}$. At all times at least one of these clauses must be TRUE. If we represent the selection criteria by 'cic_{a_i}', we can write:

$$\left(\bigOmega_{i=2}^{m-1} cic_{a_i} = \text{TRUE}\right),$$

where Ω indicates 'conjunction'.
If two or more of the clauses are TRUE simultaneously the selection between those two is indeterministic.
Section a_1 is the initial section of the structure, section a_m is the terminal section, and sections $a_2, \ldots, a_{m-1}$ are called the selection branches. In figure 4.9 a graphical representation of the selection structure is given.

(3) **Iteration**
An *iteration structure* is an ordered set of sections:

$$\{a_1, a_2, a_3, a_4\},$$

with initial state:

$$(I(a_1) = T(a_1) = I(a_4) = T(a_4)) \text{ AND } (I(a_2) = T(a_2) = I(a_3) + 1 =$$
$$T(a_3) + 1) \text{ AND } (T(a_1) + T(a_3) = I(a_2) = T(a_2) = I(a_3) + I(a_4)),$$

and the following sequence relations:

$$sic_{a_2} = (T(a_1) + T(a_3) - I(a_2) \geqslant 1)$$

$$sic_{a_3} = (T(a_2) - (I(a_3) + I(a_4)) \geqslant 1)$$

$$sic_{a_4} = (T(a_2) - (I(a_3) + I(a_4)) \geqslant 1).$$

The iteration criterion is specified in clauses for sections a_3 and a_4. The two clauses must be complementary:

$$cic_{a_3} = \text{NOT } cic_{a_4}.$$

A graphical representation of the iteration structure is given in figure 4.10.

(4) **Parallel path**
A *parallel path* is an ordered set of sections:

$$\{a_1, a_2, \ldots, a_m\},$$

with initial state:

$$(I(a_1) = T(a_1) = I(a_2) = \ldots = T(a_m)),$$

and sequence clauses:

$$(\forall\ i)\ ((1 < i < m) \rightarrow sic_{a_i} = (T(a_1) - i(a_i) \geqslant 1))\ \text{AND}$$
$$sic_{a_m} = \Delta_{j=2}^{m-1} (T(a_j) - I(a_m) \geqslant 1),$$

where Δ indicates 'disjunction'.
In figure 4.11 a graphical representation of the parallel path is given. The parallelism is visualized by a semi-circle on the relevant arcs.

Properties
All types of basic structures can be made cyclical by making sic_{a_1} = $(T(a_m) - I(a_1) = 0)$. We further state the following explicit properties of the basic structures discussed here:

(a) When a basic structure is initiated in the proper initial state, no section in that structure can be blocked forever by the clauses specified.
(b) If the structure is initiated once (with ic_{a_1} = TRUE):
    1. All sections $a_1, \ldots, a_m$ are enabled and executed precisely once, in the order in which they are listed (processing line), or
    2. section a_1 is executed, followed by one execution of one of the sections $a_2, \ldots, a_{m-1}$ followed by one execution of a_m (selection), or
    3. sections a_1 and a_2 are executed once, in this order, followed by zero or more executions of the sequence $a_3; a_2$, until cic_{a_4} is TRUE after a termination of a_2. Then a_4 is executed once (iteration), or
    4. section a_1 executes once, after which all sections $a_2, \ldots, a_{m-1}$ are enabled and executed once (random order). After completion of all sections $a_2, \ldots, a_{m-1}$ section a_m executes once (parallel path).
(c) For each single initiation of section a_1, section a_m executes precisely once, within finite time (provided the iteration terminates in iteration structures) and the completion of a_m restores the initial state. Each of the basic structures is therefore equivalent to an atomic section A, where I(A) corresponds to $I(a_1)$ and T(A) corresponds to $T(a_m)$.

We prove the three properties for processing lines and processing cycles. The proofs for the other structures are similar.

Processing lines
First we will show that the following condition is invariant to the execution of sections in a processing line:

$$(\forall\ i)\ ((1 \leqslant i \leqslant m - 1) \rightarrow (T(a_i) \geqslant I(a_{i+1}))).$$

The proof is by contradiction. Assume that $(T(a_i) < I(a_{i+1}))$ for some i. All gate variable values are non-decreasing functions of time. Initially further $T(a_i) = I(a_{i+1})$. So section a_{i+1} must have been initiated at least once while $T(a_i) \leqslant I(a_{i+1})$. But the initiation condition of section a_{i+1} contains the clause $(T(a_i) - I(a_{i+1}) \geqslant 1)$. Using axiom A(1) we can derive a contra-

diction.

We now prove that section a_i from the processing line cannot be blocked indefinitely. The blocking condition for a_i is:

$$(T(a_{i-1}) - I(a_i) < 1), \text{ for all } 1 < i \leqslant m.$$

Using the property proved above we conclude that this implies:

$$(T(a_{i-1}) = I(a_i)).$$

As terminations cannot be blocked (axiom A(3)) this implies that section a_{i-1} is blocked indefinitely.
In at most m steps this reduces trivially to the condition that ic_{a_1} = FALSE permanently, which implies that the processing line can only be blocked via the initiation condition of the entire line, but not via the internal sequence clauses.
The proof of property b is trivial. In the initial state the initiation conditions of all sections $a_2, \ldots, a_m$ are FALSE. Only the truth of ic_{a_1} can initiate the line. The termination of any section a_j ($j = 1, 2, \ldots, m - 1$) increases $T(a_j)$ by one, which enables the initiation of the succeeding section a_{j+1}. On completion all sections in the line have been executed as many times as a_1 was initiated, and all gate variables have been increased by precisely that amount, so that the initial state is restored, which proves also property c.

Processing cycle
Blocking of any section a_i from the cycle implies blocking of the initial section a_1, as was shown for processing lines above. Indefinite blocking of section a_1 implies:

$$(T(a_m) \neq I(a_1)).$$

It is not difficult to show that:

$$(T(a_m) \leqslant I(a_1) \leqslant T(a_m) + 1),$$

which implies that:

$$(T(a_m) + 1 = I(a_1)) \text{ permanently.}$$

The latter implies deadlock of section a_m, that is:

$$T(a_{m-1}) = I(a_m).$$

In precisely m steps we can then derive that:

$$(T(a_1) = I(a_2) = T(a_2) = I(a_3) = \ldots = I(a_m)).$$

Within finite time:

$$I(a_1) = T(a_1) \text{ and } I(a_m) = T(a_m)$$

This implies that: $T(a_m) = I(a_1)$.

The latter conflicts with the earlier result that:

$$(T(a_m) + 1 = I(a_1)) \text{ permanently,}$$

and thus indefinite blocking of any section in the cycle is impossible.

Structured sections

On the lowest level in a nesting hierarchy we find the atomic sections, as defined in section 4.3.1. We can define new hierarchical levels in a nesting hierarchy as follows:

(1) A hierarchically structured section of level 1 consists by definition of atomic sections, combined in one of the basic sequencing structures.
(2) A hierarchically structured section of level n consists by definition of atomic sections and hierarchically structured sections of level n-1 or lower (but at least one of level n-1), combined in one of the basic sequencing structures.

With an inductive proof one can show that a hierarchically structured section, thus defined, containing on each level only non-cyclic structures, is equivalent to an atomic section, provided that all iterations (on each level) terminate properly.

4.7. SYSTEMATIC DESIGN OF COORDINATION SCHEMES

In this chapter we have described the coordination problems in terms of formal relations (dependences) between sections. The design problem is clearly to determine a correct characterization of the coordination requirements in such relations, when given an informal problem statement.
In this section 4.7 we describe a design method which consists of 4 steps: state characterization, derivation of state invariants, rule forming, and correctness analysis.

4.7.1. Informal problem statement

We start with an informal, and perhaps ambiguous description of a coordination problem. The following example will be elaborated here:

> Three groups of processes named I, II, and III, share a data base. No more than N members of the first group and/or no more than M members of the second group may execute simultaneously in that data base. Members of the first or second group may never execute simultaneously with members of the third group. Only one member of the third group may execute at a time. No group may be starved. No group may be able to monopolize the data base.

Some of the coordination rules may already be recognized from the informal description. We will however follow another procedure here, in order to facilitate the correctness analysis of the solution.
First, the problem statement must be formalized. All ambiguity must be removed. We do this by choosing and naming sections, and by distinguishing between legal and illegal system states.
The coordination rules for all processes from the first group (and also for those from the second and third group) are all equal. In the terminology of section 4.3.2.6, these groups can be formalized as classes of processes. For simplicity we assume that the processes in the same class execute the same code. (The restriction is not essential though.) We then distinguish between three sections, which we call A, B, and C, respectively.

4.7.2. State characterization

The following states can be recognized as illegal:

- The number of processes executing section A is larger than N.
- The number of processes executing section B is larger than M.
- The number of processes executing section C is larger than 1.
- Section A is being executed simultaneously with section C.
- Section B is being executed simultaneously with section C.

The last requirement from the informal problem statement is ambiguous. A possible formalization would be:

- While processes are delayed by a request for the execution of section C section A or B may only be initiated if the 'last terminating process' executed section C. And:
- While processes are delayed by a request for the execution of either section A or B, section C may only be initiated if the last terminating process executed section A or B.

We must now choose a set of state variables which can be used to describe all illegal system states. We choose the following set:

- AA the number of processes executing section A. (An abbreviation of $I_A - T_A$.)
- AB the number of processes executing section B.
- AC the number of processes executing section C.

- WA the number of processes delayed on their request for the initiation of section A. (An abbreviation of $I_{WA} - I_A$, where WA is the name of a section which contains A.)
- WB the number of processes delayed by a request for the execution of section B.
- WC the number of processes delayed by a request for the execution of section C.
- LP the identity of the section executed by the last terminating process. (LP is a communal variable, which is accessed in sections A, B, and C.)
- AV1 the number of processes which initiated section A or B, while WC > 0 and LP = A ∪ B. (AV1 is an 'auxiliary variable', which is used only in the design and analysis phase, but which need not be implemented as such.)
- AV2 the number of processes which initiated section C, while WA + WB > 0 and LP = C.

4.7.3. State invariants

We can now determine the set of state invariants which identify the set of legal system states, by merely formalizing the complements of the illegal states given in section 4.7.2 with the aid of the state variables defined. We thus arrive at the following 3 state invariants:

(1) AA ⩽ N AND AB ⩽ M AND AC ⩽ 1;
(2) AA + AB = 0 OR AC = 0;
(3) AV1 = 0 AND AV2 = 0.

The trivial invariants, like AA ⩾ 0 or LP = A OR B OR C, are not included. (They may be considered in a more detailed correctness analysis though, in which one also considers whether the state variables themselves are correctly implemented.)

4.7.4. Rule forming

To form the coordination rules, the effects of the initiation and termination of each section in the system on the truth of the system invariants must be examined. We can make the following system-table:

		Effect	(Invariants) (1)	(2)	(3)
Initiation of	WA	WA := WA + 1			-
	WB	WB := WB + 1			-
	WC	WC := WC + 1			-
	A	AA := AA + 1	x	x	x
	B	AB := AB + 1	x	x	x
	C	AC := AC + 1	x	x	x
Termination of	A	AA := AA + 1	+	+	
	B	AB := AB + 1	+	+	
	C	AC := AC + 1	+	+	
	WA	WA := WA - 1			+
	WB	WB := WB - 1			+
	WC	WC := WC - 1			+
Access of LP in	A	LP := A			+/-
	B	LP := B			+/-
	C	LP := C			-/+

x stands for a potential violation of the corresponding invariant;
+ stands for a positive effect (can remove a pre-condition for a violation);
- stands for a negative effect (can introduce a pre-condition for violations).

Observe that the terminations do not cause violations of system invariants, and have no negative effects on these. The only potential violations occur in the initiation of the sections A, B, and C, as indeed they should.
To determine the initiation conditions we must examine under what conditions the initiation of a section causes the violation of an invariant. Using the system table we find that the initiation of section A violates invariant 1 iff AA = N. It violates invariant 2 iff AC > 0. It violates invariant 3 iff WC > 0 AND LP = A OR B.
The initiation condition for section A can therefore be written as:

$$AA < N \text{ AND } AC = 0 \text{ AND } (WC = 0 \text{ OR } LP = C).$$

This initiation condition can be interpreted as a state-transition constraint. The use of communal variable LP makes the coordination rule expressed here a C-rule. Similarly, we find the initiation conditions for sections B and C:

$$ic_B = (AB < M \text{ AND } AC = 0 \text{ AND } (WC = 0 \text{ OR } LP = C))$$

$$ic_C = (AC = 0 \text{ AND } AA + AB = 0 \text{ AND } (WA + WB = 0 \text{ OR } LP = A \cup B)).$$

For the processes in group I we thus arrive at the following code:

```
SECTION(WA);
     SECTION(A);
          RULES: (I(A) - T(A) < N) AND (I(C) - T(C) = 0) AND
                 (I(WC) - I(C) = 0 OR LP = C);
          begin
               code for group I;
               EFFECT(LP := A);
          end
     ENDSECTION(A)
ENDSECTION(WA).
```

4.7.5. Correctness analysis

To verify the design phase we must at least check the observance of the two design rules, argumented in section 4.5. We may further check the equivalence between (1) the informal problem statement, and (2) the working of the derived solution. We must then verify the consistency of the coordination rules and the reachability of desirable system states. (Cf. **Robinson & Holt '73**). As an example we consider the equivalence problem, reachability aspects and consistency.

Equivalence of descriptions
We would like to verify that the coordination rules do correctly represent and solve the problem as it was originally phrased. The rules derived should then be both *sufficient and* each clause in those rules must be *necessary* to guarantee this.
Assume the rules are not sufficient. In that case at least one of the illegal system states must be reachable. Consider the illegal state in which AA is larger than N. The initiation of a section is indivisible in the section model, so at some point section A must have been initiated when AA = N. But, clearly the initiation condition of section A would have prevented an initiation in that state as it contains the clause AA < N q.e.d. Similarly, all illegal states can be inspected, and if none of these is reachable we may conclude that the rules are indeed sufficient to solve the problem. To verify whether each clause is also necessary, let us assume that one specific clause is redundant. We may then show that deleting the clause from the condition makes at least one

illegal state reachable.

Reachability of desirable system states
On a high level of abstraction we may formulate the following five desirable system states:

1. empty state: AA = AB = AC = 0;
2. execution of section A: $0 < AA \leq N$ AND AB = AC = 0;
3. execution of section B: $0 < AB \leq M$ AND AA = AC = 0;
4. execution of section C: AC = 1 AND AA = AB = 0;
5. execution of both sections A and B: $0 < AA \leq N$ AND $0 < AB \leq M$ AND AC = 0.

State 1 is chosen as the initial state. It is readily seen that states 2, 3, and 4 are only unreachable if the corresponding sections are blocked. (We consider undesirable blockings below.) Let us consider whether state 5 can be blocked. This would imply that either section B cannot be initiated in state 2, or that section A cannot be initiated in state 3. This implies:

$$AC = 1 \text{ OR } (WC > 0 \text{ AND } LP = A \cup B).$$

Clearly, AC = 1 is impossible, as this would imply the reachability of an illegal system state. The other condition can however indeed be true each time that section B is about to be initiated in state 2, or section A is to be initiated in state 3. The conclusion must therefore be that state 5 is potentially unreachable. (That is: under some unfavorable circumstances state 5 may never be reached.)

Consistency of the coordination rules
The coordination graph for the derived solution is given in figure 4.12. We consider the possibility of an undesirable blocking of section A. The initiation condition of A contains three clauses, which we will examine below.

1. $AA < N$
   This clause cannot be FALSE permanently, as we assumed that all further initiations of A are blocked, and that terminations are never blocked. If the clause would be FALSE at some time, it must become TRUE within finite time.

2. AC = 0
   The initiation condition of section C contains the clause AC = 0. This implies that when AC is larger than zero at one time, all further initiations of C are blocked until AC = 0 is restored. If the clause is FALSE it will become TRUE within finite time.

3. (WC = 0 OR LP = C)
   The blocking condition corresponding to this third clause is: $WC \neq 0$ AND $LP \neq C$ permanently. If LP is to remain A or B, section C must clearly be blocked. (Observe that there are processes waiting for section C to be initiated, as $WC \neq 0$.) The blocking condition for section C, however, is:

   $$AC \neq 0 \text{ OR } AA + AB \neq 0 \text{ OR } (WA + WB \neq 0 \text{ AND } LP = C).$$

   The starting point of our argument was that section A is blocked, so the first clause of this condition cannot be TRUE permanently. The last clause cannot be TRUE either as it contradicts the assumption that the third clause in the blocking condition of section A is TRUE, implying that $LP \neq C$. This leaves the possibility that $AA + AB \neq 0$, which reduces (as A is blocked) to $AB \neq 0$. The latter condition can only be TRUE if section B is not blocked. The initiation condition of section B is:

(AB < M) AND (AC = 0) AND (WC = 0 OR LP = C).

The last clause of this condition must however be FALSE as it equals the complement of the third clause in the blocking condition of section A, which is examined here. If A is blocked on the third clause of its initiation condition, then B is blocked on that clause, which contradicts our argument.

Section A can therefore not be blocked on any of the three clauses from its initiation condition. It cannot be blocked on alternating blocks of these clauses, as two of these clauses (the first and the second) will always be TRUE within finite time, (see above). Section A can therefore not be blocked indefinitely at all. The arguments for sections B and C are similar.

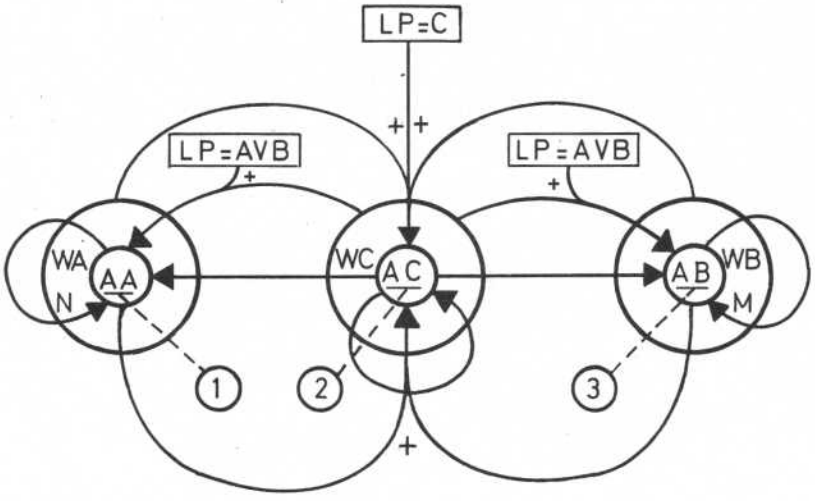


Figure 4.12.
General Graph with Blocking Conditions
EFFECT clauses:
(1) (LP := A)
(2) (LP := C)
(3) (LP := B)

4.8. TRANSLATION TO D-SEMAPHORE IMPLEMENTATIONS

As an example of the systematic transformation of a coordination scheme, described in the language of the section model, into an implementation with the extended D-semaphore operations as defined in chapter 1, we will study the solution to the third version of the readers/writers' problem. (See e.g. section 4.3.2.5 and 4.3.3.1.) We have:

$$ic_{AR} = (I(AW) - T(AW) = 0) \text{ AND } (I(WW) - I(AW) = 0 \text{ OR } PREF = TRUE).$$

Only the initiation of section AW, that is: the increasing of I(AW), can make the first clause FALSE. Only the increasing of T(AW) can make it TRUE. Similarly, only the increasing of I(WW) or the access of PREF can make the second clause FALSE, and only the increasing of I(AW) or the access of PREF can make it TRUE again. We can model these effects conveniently in semaphore UP and DOWN operations. Note that I(AW) can make both the first clause FALSE and the second clause TRUE, so we need at least two semaphore variables to represent this effect. We can represent the value of (I(AW) - T(AW)) in one semaphore D1, the value of (I(WW) - I(AW)) in one sempahore D2, and the value of PREF in one semaphore D3. Effects which can make a clause TRUE are modelled as UP operations, and their complements as DOWN operations.
The description of the readers/writers' problem, derived in section 4.3.2.5 can then be translated systematically as follows:

down D5;	-	represents the negative effect of I(WR) on ic_{AW};
(D1 ∩ (D2 ∪ D3)):	-	represents the initiation condition of AR;
down D4 **up** D5;	-	represents the negative effect of I(AR) on one clause in ic_{AW}, and its positive effect on another clause in ic_{AW};
read;	-	the code of the reader processes;
set (D3,-1; D6,0);	-	represents the updating of communal variable PREF;
up D4;	-	represents the positive effect of T(AR) on ic_{AW}.

The full translation yields the following algorithms:

```
                    down D5;
(D1 ∩ (D2 ∪ D3)): down D4 up D5;
                    read;
                    set (D3,-1; D6,0);
                    up D4;
```

```
                    down D2;
(D4 ∩ D1 ∩ (D5 ∪ D6)):down D1 up D2;
                    write;
                    set (D3,0; D6,-1);
                    up D1;
```

A further example of a systematic translation of a section description in an implementation with D-semaphores is given in section 6.7. In that section we give a proof of equivalence for the descriptions in the section model and the implementations thus derived.

4.9. AUTOMATED ANALYSIS OF COORDINATION SCHEMES

When given the set of relevant state variables (gate variables, communal variables) and their initial states it is a trivial, though perhaps lengthy, task to construct a state transition diagram, which contains all reachable system states. This state diagram can be analyzed for the presence or absence of respectively legal and illegal system states. Deadlock states can be recognized immediately, as no state transition originates in those states. Starvation loops may be harder to detect, but in principle they should be recognizable. The main difficulty of a computerized analysis of this type would be the size of the state diagram. One may attempt to reduce the diagram by using a notion of state equivalence, but then there is the danger of the introduction of spurious paths in the reduced diagram, as noted by **Howard & Alexander '73**. With a careful definition of state equivalence one should however be able to reduce a diagram to a realistic size, while preserving all relevant information for a correctness analysis. Preferably one should apply the reductions while constructing the diagram. (And not first construct a full diagram which is then reduced.)

We have developed an analysis method of this type, based on descriptions of coordination schemes with the aid of an extended type of the transition tables known from circuit design theories (the Mealy and Moore tables). With each section we then associate a transition table which models its working. The interaction between processes executing the tables is modelled by the overlapping of input and output sets of transition tables. It would consume too much paper to include a full description of this method in the present dissertation. A brief outline is included in appendix D.

4.10. REFERENCES

Ashcroft, E. & Manna, Z. (1970), *Formalization of properties of parallel programs*, Artificial Intell. Project, Memo AIM-110, Computer Science Department, Stanford University, California, U.S.A., 1970.

Atkinson, R. & Hewitt, C. (1975), *Synchronization in actor systems*, Working paper ACM SIGCOMM/SIGOPS Interprocess Communications Workshop, Santa Monica, California, U.S.A., March 1975, *Revised* December 1975, 13 pgs.

Belpaire, G. (1975), *On programming dependences between parallel processes*, Techn. Report 244, Department of Computer Sciences, University of Wisconsin, Madison, U.S.A., (Ph.D. Thesis, University Leuven, Belgium), 1975.

Brinch Hansen, P. (1973), *Operating system principles*, Prentice Hall, Englewood Cliffs, N.J., U.S.A., 1973.

Denning, P.J. (1971), *Third Generation of computer systems*, Computing Surveys, Vol. 3, No. 4, Dec. 1971, 175-216.

Dijkstra, E.W. (1975), *Guarded commands, non-determinancy and formal derivation of programs*, Comm. ACM, Vol. 18, No. 8, Aug. 1975, 453-457.

Hoare, C.A.R. (1974), *Monitors, an operating system structuring concept*, Comm. ACM, Vol. 17, No. 10, Oct. 1974, 549-557. (Errata in Vol. 18, No. 2, Febr. 1975.)

Horning, J.J. & Randell, B. (1973), *Process structuring*, Computing Surveys, Vol. 5, No. 1, 1973, 5-31.

Howard, J.H. & Alexander, W.P. (1973), *Analyzing sequences of operations performed by programs*, In: *Program Test Methods*, W.C. Hetzel (Ed.), 1973, 239-254.

Lauer, P.E. & Campbell, R.H. (1975), *Formal semantics of a class of high level primitives for coordinating concurrent processes*, Acta Informatica, Vol. 5, No. 4, 1975, 297-332.

Müller, K.G. (1976), *On the feasibility of concurrent garbage collection*, Ph.D. Thesis, Univ. of Techn. Delft (The Netherlands), 1976, 193 pgs.

Robinson, L. & Holt, R.C. (1973), *Formal specifications for solutions to synchronization problems*, SRI Report, Stanford Research Institute, Computer Science Group, Menlo Park, California, U.S.A., November 1973.

Sintzoff, M. & Van Lamsweerde, A. (1974), *Constructing correct and efficient programs*, MBLE Research Lab Report R266, September 1974, (Cf. *ref.* pg.**129**).

Wirth, N. (1969), *On multiprogramming, machine-coding and computer organization*, Comm. ACM, Vol. 12, No. 9, Sept. 1969, 489-498.