Computing Science Technical Report No. 87

The Design of Coordination Schemes

Gerard J. Holzmann

August 7, 1980

The Design of Coordination Schemes

Gerard J. Holzmann

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

A model of a multiprocessing system is introduced that allows us to design, analyze and implement coordination schemes in a stepwise manner. A distinction is made between two principal design phases: 1) the design of a consistent set of coordination rules, and 2) the design of a minimal and complete signaling scheme. The correctness of a design can be established independently for both phases.

The working of the model is based on the existence of a hypothetical machine called a *guard*. The restrictions implied by the idealized properties of this guard machine are removed in later design phases. Meanwhile, the same restrictions allow for straightforward correctness analysis, of which the validity can be shown to be preserved by the later refinements.

The model allows one to derive schemes which are largely implementation independent. It is illustrated how specific implementations in Concurrent Pascal can be generated mechanically by a translator program.

In an appendix a new set of D-semaphore operations is introduced that allows for a convenient short-hand description of coordination schemes.

August 7, 1980

An earlier version of this paper was presented at the 'Workshop on Concurrency', USC, Idyllwild, California, March 13-16, 1980.

Part of this research was supported by the Netherlands Organization for the Advancement of Pure Research (ZWO, The Hague, Netherlands).

The Design of Coordination Schemes

Gerard J. Holzmann

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

In the past 10-15 years many methods have been suggested for the specification and analysis of coordinations in concurrent programs [10,15]. The methods suggested for the specification of coordinations mostly involve the use of special programming primitives, to be executed as indivisible operations. These tools evolved from the simple lock/unlock and test & set statements, and their many equivalents, via the important semaphore concept [7], to the structured monitors developed by Brinch Hansen and by Hoare [9], and the high-level path expressions of Campbell & Habermann [5].

The early tools were understood and implemented directly in terms of machine instructions, for instance with interrupt masking techniques, or with the aid of special hardware arbiter circuits. The more recent tools can be implemented in layers, with the aid of earlier tools. There are, however, only few methods that allow for an implementation-independent specification of coordinations [8,10,17].

For the analysis of the effects of a coordination one can choose from a range of graph methods and token nets [3,10], plus a relatively small number of theorems in first-order predicate calculus [13].

The graphs and token nets are mostly derived from respectively the Slutz graph [18] and the Petri Net model [14]. Both the specification tools and the analysis methods are gradually being defined on higher levels of abstraction. The more recent tools allow for a more extensive and more explicit use of state variables (sometimes called *history variables*, or *auxiliary variables*), as this not only clarifies descriptions, but can also simplify their analysis [11,13].

Still, even with the most structured tools it can be quite difficult to represent the intent and working of a coordination in a conceptually clear way, and it is not much easier to analyze the correct working of a coordination convincingly on such items as freedom of deadlock and starvation. With the method to be described below some of these difficulties can be overcome.

Much in line with the developments outlined above, this method is based on systematic abstraction and an extensive use of state variables. The method allows one to design and analyze coordination schemes in a stepwise manner.

In the first phase, one abstracts entirely from all computation, signaling and queueing aspects, and one concentrates merely on the formulation of a consistent set of ordering constraints for abstract blocks of program code. These ordering constraints can be analyzed on their consistency in a fairly straightforward manner, as a result of the high level of abstraction at which they are formulated.

In a second phase, the constraints are made more specific, and the signaling and queueing aspects are included to optimize the scheme. In a final stage, the actual computations to be protected are elaborated, and analyzed with more standard verification techniques (e.g. with an inductive assertion method). The descriptions can then be implemented with almost any type of coordination primitive. In many cases the descriptions can be 'translated' directly into clear structures with D-operations [1,2,19]. All descriptions can be translated (even mechanically) into standard monitor structures.

Comparable methods for the description and analysis of coordination structures were elaborated by Robinson & Holt [17], Belpaire [1], and Keller [12].

First we will describe the conceptual model on which the method is based. We will then give some

examples of descriptions in the language of that model, and we will show how these descriptions can be analyzed. Finally, we will illustrate how the descriptions can be refined and implemented.

2. Conceptual model

2.1. Coordinated systems and coordination schemes

For the description of coordination constraints on a high level of abstraction we will use a model of a multiprocessing system. The model is intended as a conceptual tool which will allow us to give a first formalization of coordination requirements.

In this model we define a *coordinated system* as consisting of:

[1] A finite collection of *sections*,

[2] An abstract machine called a *guard*, and

[3] A finite collection of *user processes*.

The sections contain the program code that is to be protected by the coordination; the user processes will attempt to execute that code; the guard will enforce the desired coordination.

Each section consists of three parts:

[1] A *head*,

[2] A *body*, and

[3] A *tail*.

In the section heading we specify the name of the section and an *initiation condition*. A section body contains either a sub-section or a program. The latter may be of any size: it may be empty, it may consist of a single operation, or it may contain an entire parallel program; it may, however, not contain other sections. Sections may not overlap each other[1].

A section which contains no sub-sections is called an *atomic section*. The tail of a section is, for the moment, merely a formalized closing statement in which the name of the section is repeated.

Coordination constraints will be enforced via the initiation conditions. A set of initiation conditions for the sections of a coordinated system is called a *coordination scheme*. The initiation conditions themselves are functions of general state variables, as will be illustrated below in number[2] 2.2.

The guard controls the access of user processes to the sections in a coordinated system. A *process* is interpreted here as an abstract entity which can request permission from the guard to access a section, await the receipt of that permission, and execute the section body. When the execution of the section body is completed, the process is assumed to notify the guard of this event. A process thus symbolizes an *actor* in this model, rather than an *action* or a computation.

The guard works on a list of initiation conditions and a finite set of state variables. Initially the guard's list is empty. A user process can induce the guard to enter a condition into the list by making a request for permission to access a specific section, specifying the section name and the corresponding initiation condition. Whenever the guard's list is non-empty the guard will evaluate the conditions on that list one by one, in order of arrival. When the guard encounters a true condition, he will release the process that caused the guard to include that specific condition in the list, and he will update the state variables accordingly. The condition is then removed from the guard's list. The released process can now access the section corresponding to the condition evaluated by the guard.

The evaluation process as a whole cannot be interrupted. It terminates only if *none* of the conditions in the list is found to be true in a complete scan of the list. A scan always starts at the top of the list. Every change in the state variables triggers a new scan, at the earliest possible moment.

Note that only the guard can change the state variables, and only in reaction to user requests or

1. It is not difficult to relax some of these restrictions, but it hardly ever seems to be necessary.
2. To avoid confusion we will use the term *number* to indicate the sections of this paper.

notifications.

When the guard has allowed a process to access a section, has updated the relevant state variables and removed the condition in question from the list, the section mentioned is said to be initiated. When a process completes the execution of a section, by executing the section tail, it will notify the guard. The guard will then update the state variables, and allow the process to proceed without delay. When these actions are completed the section is said to be terminated.

Observe that although any number of user processes can be executing simultaneously, the evaluation of initiation conditions and the initiation and termination of sections are executed strictly one-at-a-time, in *exclusive mode*.

Note also that more than one process can request permission to access the same section. The guard will then enter one condition in the list for each request received, each condition being associated with another process (i.e. the identity of the requester). One process can, however, only request access to one section at a time. The number of processes is assumed to be finite, but is not bounded in any way.

A process is said to be *blocked* from the moment that the guard evaluates the initiation condition of a section to which it has requested access and finds it false, to the moment that the guard finds the condition to be true and permits the process to proceed. Note that a process is not considered to have been blocked if the condition is found to be true in the first evaluation by the guard.

The working of a coordinated system can be formalized in the following four axioms[3]:

A(1) A section can only be initiated if the corresponding initiation condition is true.

A(2) A process can only be blocked in a section heading. An atomic section will always terminate within a finite time after it was initiated.

A(3) If the initiation condition of a section becomes true infinitely often, all processes blocked in the corresponding section heading will be released within a finite time.

A(4) The evaluation of an initiation condition plus - if the condition is found to be true - the corresponding initiation can be considered as one indivisible act, which is mutually excluded from all other accesses of the state variables involved.

Observe that we concentrate on coordination aspects here, and not on computations. At this stage, we do not concern ourselves with the blocking of processes in non-terminating loops within section bodies, but only with blockings which are the result of the enforcement of the coordination constraints (cf. axiom A (2)). Notice also that an atomic section will always terminate as many times as it was initiated. The same is not necessarily true for non-atomic sections.

2.2. State variables

Each section is assigned a unique name within the coordinated system. With each section we further associate two abstract gates: an entry gate and an exit gate. The state of these two gates is expressed in *gate variables* (comparable to Keller's *place variables* [12]). The term *gate variable* stems from Robinson & Holt [17].

The gate variables for the entry gate are:

I(A) which indicates the number of times that section A has been initiated since system initialization, and

W(A)which indicates the number of processes that are blocked in the heading of section A[4].

The gate variable for the exit gate is:

T(A) which indicates the number of times that section A has been terminated since system

3. It is tempting to formalize these axioms in temporal logic. For the purpose of this paper, however, the informal version given here is preferred.

4. The use of gate variable W(A) can be avoided completely. It is introduced to allow for a more concise description of the coordinated systems.

initialization.

We shall see below that the gate variables play an important role in the analysis of a coordination scheme.

At system initialization we have, by definition, for all sections:

$$I\ (\text{name}) = W\ (\text{name}) = T\ (\text{name}) = 0. \tag{2.2.1}$$

As each section can only be terminated if it was first initiated, we have for all sections, invariantly:

$$I\ (\text{name}) \geq T\ (\text{name}) \geq 0 \tag{2.2.2}$$

and trivially, also:

$$W\ (\text{name}) \geq 0 \tag{2.2.3}$$

at each time after system initialization.

The value of the expression

$$(I(name) - T(name))$$

indicates the number of processes executing the corresponding section.

The values of the gate variables are clearly potentially unbounded. This, however, need not worry us here as these variables are merely used as a formalization of an abstract notion, which can help us in the design of a lucid coordination scheme. The place and the use of gate variables is therefore quite different from that of eventcounts [16]. The gate variables need not play a role in actual implementations, as shall be shown below.

With only gate variables, as defined above, we can already describe many types of coordination, e.g. exclusions and simple orderings. For instance, we can limit the number of processes that can execute section A simultaneously to any positive number N by defining the following initiation condition for A (abbreviated hereafter as IC(A)):

$$IC(A) = (I(A) - T(A) < N).$$

In fact, we can define any type of partial ordering on the executions of the sections in this manner.

To increase the power of the method further we define a second type of state variable, which we shall call the *communal variable*. Communal variables may occur in initiation conditions together with the gate variables. The important difference with the gate variables is, however, that the communal variables can be accessed and updated more explicitly. The precise way in which the communal variables are accessed can be specified in special *effect clauses*. For simplicity we require that effect clauses only occur in a section heading and/or in a section tail, but not scattered through the program code in section bodies. An effect clause which is placed in a section heading is called an *initiation effect* (abbreviated: IE); an effect clause which is placed in a section tail is called a *termination effect* (abbreviated: TE). The introduction of the effect clauses yields one more axiom:

A(5) The initiation of a section and the execution of the initiation effect (if specified) can be considered as one indivisible act which is mutually excluded from all other accesses of the state variables involved. The same is true for terminations and termination effects.

As we shall see, the fact that the communal variables, unlike the gate variables, can be updated in arbitrary ways (within effect clauses) makes them less suitable for use in correctness arguments. We will use the following notation for coordinated systems:

```
SYSTEM systemname;
declaration of communal data;

SECTION sectionname (formal parameters);
declaration of data local to the section;
  head;
  body;
```

```
  tail;
END sectionname

{/ declaration of other sections};

initialization of communal data;
END systemname;
```

The details of this notation will be illustrated with examples in the numbers 2.3 and 4.1 to 4.4 below.

2.3. Introductory example

As a first example of the application of this design method to a coordination problem we consider a simple version of the readers and writers' problem [6].

> R readers and W writers share a data base. The coordination must be such that only one writer can access the data base at a time, or any number of readers simultaneously. Reading is considered to be of more importance than writing, so a 'stream' of executing readers may not be interrupted by the writers.

We define two sections: one for reading, named AR, and one for writing, named AW. The initiation conditions for these two sections are readily found:

$$IC(AR) = (I(AW) - T(AW) = 0), \text{ and}$$

$$IC(AW) = (I(AW) - T(AW) = 0) \cap (I(AR) - T(AR) = 0).$$

The corresponding system is represented in the following structure:

```
SYSTEM readers&writers1;

SECTION AR;
  IC: [I(AW) − T(AW) = 0];
  "read";
END AR /

SECTION AW;
  IC: [I(AW) − T(AW) = 0  ∩  I(AR) − T(AR) = 0];
  "write";
END AW;

END readers&writers1;
```

There are no initiation or termination effects specified in this description. The place to do so, in more complex cases, would be directly after the initiation condition and/or directly before the closing statement of the section, respectively.

We use the symbol "/" to separate the sections (it implies no ordering). Both sections are atomic sections. Observe that the computations themselves remain uninterpreted. The updatings of the gate variables by the guard (e.g. the incrementing of I(AW) at the initiation of section AW, and the incrementing of T(AW) at its termination) are left implicit.

For the relatively simple problem considered here the initiation conditions IC(AR) and IC(AW) can also be derived more formally from a *system invariant:*

$$((I(AR) - T(AR) = 0) \cup (I(AW) - T(AW) = 0)) \cap (I(AW) - T(AW) \leq 1).$$

The initiation conditions are the weakest pre-conditions on the execution of the section bodies that guarantee the integrity of this invariant. A system invariant of the above type formalizes the legal *states* of the data base, and it is required that all other states be unreachable.

For simple problems such a system invariant is easy to find and indeed offers a reliable first formalization of the coordination requirements. For some problems, however, as we shall discuss in the numbers 4.3 and 4.4, it is not so easy to express the coordination requirements in legal and illegal states of the system as a whole. To illustrate this, let us consider a readers and writers' problem in which the writers are to be given priority over the readers. In this case the readers should not be able to start their reading when writers are waiting. We can formalize this concisely in an initiation condition by extending IC(AR) with the clause: W(AW) = 0, so that it becomes:

$$IC(AR) = (I(AW) - T(AW) = 0 \cap W(AW) = 0).$$

If, however, we want to derive this new condition formally from a system invariant we run into serious problems. The problem is that this requirement cannot be expressed unambiguously in an illegal state of the gate variables with the present tools. The legality of the state

$$(I(AR) - T(AR) > 0 \cap W(AW) > 0)$$

depends on the *execution history*. Considerations like these have led Robinson & Holt [17] to make a distinction between *static* and *non-static* invariants. The requirements from the second readers and writers' problem, as discussed here, can with their method be formalized in a non-static invariant, as follows:

$$\min (I(AR) - I'(AR)) \leq 1 - min(W(AW), 1).$$

Where I'(AR) represents the value of I(AR) directly before the most recent state transition.

Alternatively, the requirement can be expressed in the *behavior specification language* developed by Greif [8], or with the aid of path expressions [5]. For longer sequences of state changes, however, the behavioral descriptions tend to be long and complicated.

The problem encountered here is that we do not really want to define an illegal state, but an illegal transition. A formalization directly in terms of initiation conditions appears to be conceptually simpler, and in that respect more reliable. Sequences of state changes can be abstracted transparently in additional state variables, as will be illustrated below in number 4. In particular, the detour via system invariants will add no clarity and will not improve the reliability of the descriptions in these cases. In the sequel we will not use system invariants for the derivation of initiation conditions.

3. Analysis of coordination schemes

The consistency of a given coordination scheme can be analyzed straightforwardly with an informal procedure. One can study the schemes on the absence of deadlocks and starvations, the observance of required exclusions, the reachability of desirable system states, etc. As an example we will consider the possibility of a permanent block on the writer processes in the coordinated system presented in the preceding number.

The writer section AW is an atomic section. It follows (axiom A(2)) that a permanent blocking of writer initiations must always lead, within finite time, to a state in which

$$I(AW) - T(AW) = 0. \tag{3.1}$$

In that state the initiation conditions of sections AR and AW reduce effectively to

$$IC(AR) = true, \text{ and} \tag{3.2}$$

$$IC(AW) = (I(AR) - T(AR) = 0). \tag{3.3}$$

(3.2) states that there is no restriction on the execution of section AR.

Now, it is easy to see that the following is a corollary of axiom A(2):

> if A is an atomic section, the state $(I(A) - T(A) > 0)$ cannot be maintained permanently by a single process.

For writers to be blocked permanently, the readers must be able to maintain a state in which

IC(AW) = false, or

$$I(AR) - T(AR) > 0 \tag{3.4}$$

(cf. invariant (2.2.2), and axiom A(1)). The above corollary then implies that the writers can only be *starved* if there is always more than one reader active. Only then could section AR be executed in overlap, and thus condition (3.4) be maintained.

To show that readers cannot be starved by the writers we can again apply the corollary of A(2) and make use of the fact that the coordination scheme does not allow more than one writer to execute at a time.

To convince ourselves that readers and writers cannot execute concurrently we may verify, with the aid of axiom A(4), that a state in which

$$(I(AR) > T(AR)) \bigcap (I(AW) > T(AW)) \tag{3.5}$$

cannot be reached without the violation of at least one of the initiation conditions. The latter would, however, violate axiom A(1).

We will give examples of the analysis of more comprehensive schemes below.

For a successful analysis of this type it is of course necessary that indeed all coordination constraints be explicited in the initiation conditions, including the ordering constraints that may be enforced implicitly via the ordering of section calls in the program texts executed by the user processes. In practice the observance of this requirement, however, is usually trivial.

4. Further examples

We restrict ourselves to a discussion of only a few standard coordination problems here, to illustrate the design method.

4.1. The bounded-buffer problem

Two processes share the access to a bounded buffer. One process is called the *producer;* the other is called the *consumer*. The producer places portions in the buffer; the consumer removes these portions in fifo order. The capacity of the buffer is N portions.

Let P be the name of the section which contains the code for producing portions, and C the name of the section which contains the code for consuming portions. Let *pr1* be the name of a variable which the producer uses to address the buffer, and similarly, let *pr2* be the name of the corresponding variable used by the consumer.

With two communal variables, named *top* and *bottom,* we arrive at the following system. The initial value of both communal variables is specified in the last clause.

```
SYSTEM bounded-buffer;
VAR top, bottom: communal integer;

SECTION P(VAR pr1: integer);
  IC: [I(P) - T(C) < N];
  IE: [pr1 := top];
  "Produce portion in buffer (pr1)"
  TE: [top := (top MOD N) + 1];
END P /

SECTION C(VAR pr2: integer);
  IC: [T(P) - I(C) > 0];
  IE: [pr2 := bottom];
  "Consume portion from buffer(pr2)";
  TE: [bottom := (bottom MOD N) + 1];
END C;
```

INITIAL: [top := bottom := 1];
END bounded-buffer;

Observe that the users may not access the communals *top* and *bottom* in the section bodies to address the buffer.

Note also that this solution is only correct for a single producer and a single consumer process, though it can trivially be extended to suit more complex cases.

From this example we see that variables (and constants) can be passed as parameters from user processes to sections, where they can then be used, for instance, to update communals (see number 4.2), or, as in the present case, to be updated by communals.

It would certainly be interesting to consider in more detail if and how this parameter passing could violate the integrity of the coordinated systems. We will however not concern ourselves with that question here.

We will prove the following statements about the bounded buffer scheme:

a. The system cannot deadlock;
b. The producer and the consumer cannot access the same buffer place simultaneously;
c. The consumer cannot remove portions from non-filled places; and
d. The producer cannot place portions in non-emptied places.

Proof of statement a:

Let us assume that a state of deadlock would be reachable, and see whether this leads to a contradiction. In that case both IC(P) and IC(C) will be false permanently:

$$I(P) - T(C) \geq N \tag{4.1.1}$$

$$T(P) - I(C) \leq 0 \tag{4.1.2}$$

We further know, by axiom A(2), that within finite time:

$$I(P) = T(P) \tag{4.1.3}$$

$$I(C) = T(C) \tag{4.1.4}$$

Combining (4.1.1) and (4.1.3) yields

$$T(P) \geq T(C) + N \tag{4.1.5}$$

Combining (4.1.2) and (4.1.4) yields

$$T(P) \leq T(C) \tag{4.1.6}$$

Clearly (4.1.5) and (4.1.6) contradict for all values of $N > 0$, and so our assumption that the system could reach a deadlock state is disqualified.

Proof of statement b:

The violation of b. would imply that a state is reachable in which:

$$(top = bottom) \tag{4.1.7}$$

$$(I(P) - T(C) < N) \tag{4.1.8}$$

$$(T(P) - I(C) > 0) \tag{4.1.9}$$

Variable *top* is updated every time T(P) is increased, and similarly, variable *bottom* is updated whenever T(C) is increased. However, *top* and *bottom* are initialized to one, while T(P) and T(C) are initially to zero (eq. 2.1.1). The following equivalences therefore hold:

$$top = (T(P) \bmod N) + 1 \tag{4.1.10}$$

$$bottom = (T(C) \bmod N) + 1 \tag{4.1.11}$$

Combining (4.1.7), (4.1.10) and (4.1.11) yields the requirement:

$$T(P) \bmod N = T(C) \bmod N \tag{4.1.12}$$

Before proceeding with (4.1.12) we will derive another invariant which will prove helpful. First observe that

$$(I(P) - T(C) \leq N \tag{4.1.13}$$

is invariant, as the only event that could violate it after system initialization would be the initiation of section P while IC(P) is false, which is prevented by axiom A(1). Similarly, we have invariantly:

$$(T(P) - I(C)) \geq 0 \tag{4.1.14}$$

As a direct result of axiom A(2) we further also know that, invariantly:

$$I(P) \geq T(P) \tag{4.1.15}$$

$$I(C) \geq T(C) \tag{4.1.16}$$

(cf. equation (2.2.2) above.) Combining (4.1.13) and (4.1.15) yields:

$$T(P) - T(C) \leq N \tag{4.1.17}$$

Combining (4.1.14) and (4.1.16) yields:

$$T(P) - T(C) \geq 0 \tag{4.1.18}$$

Combining (4.1.17) and (4.1.18) yields:

$$O \leq (T(P) - T(C)) \leq N \tag{4.1.19}$$

Combining (4.1.19) with (4.1.12), finally, yields:

$$(T(P) - T(C) = 0) \bigcup (T(P) - T(C) = N) \tag{4.1.20}$$

Clearly, (4.1.20) contradicts (4.1.8) and (4.1.9) (combined with (4.1.15) and (4.1.16)), which proves statement b.

Proof of statement c:

At system initialization we have:

$$(top = bottom) \bigcap (I(P) = I(C) = T(C) = 0) \tag{4.1.21}$$

From IE(P) and IE(C) we can see that the producer and the consumer will access the buffer-places in the same order, from place 1 to N, repeatedly. Statements c. and d. assert that the consumer cannot overtake the producer and vice versa.

To violate c. the consumer must remove more portions than the producer adds to the buffer:

$$(I(C) > I(P)) \tag{4.1.22}$$

Combined with (4.1.15) this gives:

$$(I(C) > T(P))$$

or equivalently:

$$(T(P) - I(C) < 0) \tag{4.1.23}$$

This contradicts invariant (4.1.14) and disqualifies the assumption.

Proof of statement d:
The violation of d. would imply that at some time

$$(I(P) - I(C) > N) \tag{4.1.24}$$

combining (4.1.24) with (4.1.16) yields:

$$(I(P) - T(C) > N) \tag{4.1.25}$$

which, however, contradicts invariant (4.1.13).

4.2. Scheduled wait

User processes must be able to delay themselves for some specified number of time units N. The existence of a special clock process, initiated at each new time unit, is assumed. We describe the working of that clock process with a section called "tick", which updates a communal variable *now*.

To delay itself a user process must execute two nested sections: "wake-me (N: integer)", and "wait". Apart from variable *now* we use an integer communal variable *alarmset*, which is local to section "wake-me". The default value for a missing initiation condition is *true*.

```
SYSTEM scheduled-wait;
VAR now: communal integer;

SECTION wake-me (N: integer);
VAR alarmset: communal integer;
 IE: [alarmset := now + N];
 SECTION wait;
  IC: [now ≥ alarmset];
  skip;
 END wait;
END wake-me /

SECTION tick;
 IE: [now := now + 1];
 skip;
END tick;

INITIAL: [now := 0]
END scheduled-wait;
```

The problem here was not to protect code against unintended interference but merely to delay a

process execution. The section bodies of "wait" and "tick" are therefore empty (indicated by 'skip'). Notice that variable *now* represents the value of gate variable I(tick).

4.3. Readers and writers

We have discussed a simple version of this problem in number 2.3. In addition to the requirements phrased there we will demand that neither readers nor writers may be starved. More specifically, Hoare [9] stated that:

"...A new reader should not be permitted to start if there is a writer waiting."

"...All readers waiting at the end of a write should have priority over the next writer." (p.556)

We will use one communal integer *pref* with initial value zero.

```
SYSTEM readers&writers2;
VAR pref: communal integer;

SECTION AR;
 IC: [I(AW) − T(AW) = 0 ∩ (W(AW) = 0 ∪ pref > 0)];
 IE: [IF pref > 0 THEN pref := pref − 1];
 "read"
END AR /

SECTION AW;
 IC: [I(AW) − T(AW) = 0 ∩ I(AR) − T(AR) = 0 ∩ pref = 0];
 "write";
 TE: [IF W(AW) > 0 THEN pref := W(AR)];
END AW;

INITIAL: [pref := 0];
END readers&writers2;
```

We will consider whether the writers can be blocked permanently in this system (cf. number 3.) When writers are blocked we must have:

$$W(AW) > 0 \tag{4.3.1}$$

and within finite time (cf. (3.1)):

$$I(AW) - T(AW) = 0 \tag{4.3.2}$$

With (4.3.1) and (4.3.2) the initiation conditions for AR and AW reduce to:

$$IC(AR) = (pref > 0) \text{ and} \tag{4.3.3}$$

$$IC(AW) = (pref = 0) \cap I(AR) - T(AR) = 0) \tag{4.3.4}$$

To block writers we must have, permanently:

$$(pref > 0) \cap (I(AR) - T(AR) > 0) \tag{4.4.4}$$

We already know that the second part of (4.4.4) can remain true if we have more than one reader

process, which can occupy section AR in overlap. (4.3.3) however implies that to make this possible either ($pref > 0$) must permanently be true, or it must become true repeatedly. The latter is impossible because once *pref* is set to zero only executing writers could cause *pref* to be incremented again (via TE(AW)).

If, on the other hand, ($pref > 0$) is true at one time it will be false in precisely |pref| initiations of reader processes. Observe that readers must keep initiating section AR in order to maintain (4.4.4), which can be deduced from axiom A(2).

We have shown that there is no execution sequence for which (4.4.4) can remain true permanently. Writer processes can therefore not be blocked indefinitely by the readers.

With a similar argument it can be shown that readers cannot be starved, and that this system cannot deadlock.

5. Refinement and implementation

5.1. Implementation with D-operations

The manipulation of the gate variables can be translated fairly straightforwardly into standard D-operations as defined by Wodon [19] and Belpaire [2], and even more so if these are extended along the lines suggested in [10] (see appendix).

As an example we consider the scheme for the bounded-buffer problem, discussed in number 4.1.

We can use two D-semaphore variables to represent the following abstractions from the scheme:

$$nf = -(1 + (I(P) - T(C) - N))$$
$$ne = +(1 + (T(P) - I(C)))$$

where

nf: non-full, and *ne*: non-empty.

If, for brevity, we abstract from the manipulation of communals *top* and *bottom* we obtain the following, well-known, solution by a direct translation of conditions and effects into *passages* and semaphore *up* and *down* operations:[5]

```
COBEGIN
"producer:" REPEAT
              nf: DOWN nf;
              "produce;"
              UP ne;
            FOREVER;

"consumer:" REPEAT
              ne: DOWN ne;
              "consume;"
              UP nf;
            FOREVER
COEND.
```

All signaling and queueing actions are still implicit in this translation. The model introduced in this paper, however, allows us to 'tune' a coordination explicitly by designing an optimal signaling and queueing scheme.

Such refinements of the schemes lead, as we shall see below, most naturally to implementations with monitor structures.

5. The notation is defined in the appendix.

We will first describe the extension of the coordinated systems to what we shall call, the *refined systems*, and then apply the extensions to all schemes discussed above.

5.2. Extension of the model

In the refined systems we allow for an explicit control of queueing and signaling. For this purpose we extend the headings and tails of the sections from the coordinated systems with some additional clauses. With these additional clauses the user processes can take over the scheduling tasks from the abstract guard. The guard will therefore have no function in refined systems.

In the refined systems the section headings consist of the following five parts:

[1] The keyword SECTION followed by a unique *section-name*,

[2] The *initiation condition* (IC), with default value *true*,

[3] The *queueing part* (QP), with default *enterq(section-name)*,

[4] The *initiation effect* (IE), with default *skip*, and

[5] The *signaling part* (SP), with default *skip*.

The first parameter in the system call *enterq* identifies the queue in which the calling process is to be included. An optional second parameter specifies a queueing priority. (Queueing is fifo by default.)

The section tails consist of three parts:

[1] The *termination effect* (TE), with default *skip*.

[2] The *tail signaling part* (TP), with default *skip*, and

[3] The keyword END followed by the *section-name*.

The working of a refined system is as follows:

A process that wants to access a section body starts by evaluating, in exclusive mode, or, as we shall call it from this point on, 'with monitor exclusion', the initiation condition of that section. If the condition is false the process will execute the queueing part, release the monitor exclusion, and block. If the condition is true in the first evaluation, the process executes the initiation effect and the signaling part, in that order[6].

A process that is released from a waiting queue will also execute both the initiation effect and the signaling part. Whether the monitor exclusion is released or not in either case depends on the outcome of the signaling part, and should be specified there.

A process that terminates the execution of a section body executes the termination effect and the tail signaling part as one indivisible act (with monitor exclusion).

We use two types of primitive operations for the releasing of processes from waiting queues.

1. If the released process is allowed to continue without re-evaluating the initiation condition of the section it is attempting to access, we use the standard operation:

   continuefq(qname);

   (fq is short for 'first-of-queue'). As the signaling part is always the last item to be executed in a section heading or tail we can transfer the monitor exclusion from signaling process to released process. Clearly, this implies that only one process may be released per signaling part in this manner. The operation has no effect if the waiting queue specified is empty.

2. If the released process has to re-evaluate its initiation condition before proceeding (with the chance of being blocked anew) we use the standard operation:

   resumefq(qname);

---

6. In many cases an initiation as such cannot make another initiation condition true. The main function of the signaling part in the head of a section is for the transfer of signaling tasks from one clause to another, as shall be illustrated in number 5.3.

We will again assume that the monitor exclusion is transferred from signaling process to released process, as above, and that the operation has no effect if the queue is empty.

Initially, we will also use yet a third type of releasing operation, to simulate the working of the guard. This third operation is, however, only used in the process of refinement, and need not play a role in an actual implementation.

3. With the operation

   restart(qname);

   We indicate that all processes that are delayed in the queue specified are to be released, without a transfer of monitor exclusion. All released processes should re-evaluate their initiation conditions. The released processes will have to compete with each other to obtain the monitor exclusion. The operation has no effect if the queue is empty.

The signaling parts mostly take the following form:

IF *condition* THEN *continuefq*(*name*1) ELSE *continuefq*(*name*2);

To distinguish these signaling parts more clearly from the surrounding code we will use the following abbreviated notation:

condition → *continuefq*(*name*1) / *continuefq*(*name*2);

To simulate the working of the guard we can, initially, include the following signaling part in all section tails (cf. footnote 6):

TP = [FOR (*all queues*) DO *restart*(*queue*)];

We call this the *guard clause*, and abbreviate it as:

TP = [GUARD];

Below we shall see how the guard clause can be amended systematically to yield more efficient solutions of the schemes discussed earlier.

5.3. Refinement of the first readers and writers' scheme

In a first refinement step for the scheme given in number 2.3 we arrive at:

```
SYSTEM readers&writers1;

SECTION AR;
  IC: [I(AW) − T(AW) = 0];
  "read";
  TP: [GUARD];
END AR /

SECTION AW;
  IC: [I(AR) − T(AR) = 0 ∩ I(AW) − T(AW) = 0];
  "write";
  TP: [GUARD];
END AW;

END readers&writers1;
```

Note that the queueing parts obtain their default contents:

QP(*AR*) = [*enterq*(*AR*)]

QP(*AW*) = [*enterq*(*AW*)]

The effect clauses are empty.

To make the signaling more efficient we can make use of the knowledge we have of the state of the gate variables at the moments when the signaling parts are executed. Consider, for instance, the signaling part in the tail of section AW. This part is only executed when a writer is just completing the execution of the body of AW, which implies that the new state of the gate variables must be:

$$(I(AW) - T(AW) = 0 \cap I(AR) - T(AR) = 0)$$

This implies that both IC(AR) and IC(AW) are true. As waiting readers should take priority over waiting writers we must start by signaling all waiting readers, if any. Thus we arrive at:

$$TP(AW) = [restart(AR)];$$

If there are no waiting readers we may consider the releasing of waiting writers. Clearly, there is no need to release all writers waiting, as only one of them will be allowed to execute anyway. We can therefore extend TP(AW) as follows:

$$TP(AW) = [W(AR) > 0 \rightarrow restart\ (AR)\ /\ continuefq(AW)]$$

The simultaneous restarting of all waiting readers, necessarily without a transfer of monitor exclusion, can still be further improved upon. Note that only one reader can initiate section AR at a time, due to axiom A(4). It is therefore better to make:

$$TP(AW) = [W(AR) > 0 \rightarrow continuefq(AR)\ /\ continuefq(AW)]$$

And add in the section heading of AR:

$$SP(AR) = [continuefq(AR)]$$

Observe, finally, that only a terminating writer can remove a blocking condition for waiting readers. We therefore need not signal reader processes in TP(AR).

When a reader terminates we know that

$$(I(AW) - T(AW) = 0 \cap I(AR) - T(AR) \geq 0)$$

We need only signal writers if IC(AW) = true, which leads to:

$$TP(AR) = [I(AR) - T(AR) = 0 \rightarrow continuefq)]$$

At this point we should convince ourselves that the signaling scheme we have designed is both minimal and complete.

A signaling scheme is said to be 'complete' if every blocked process that could continue without violating an initiation condition is always signaled within finite time (as required by axiom A(3)).

A signaling scheme is called 'minimal' if no process can be signaled in vain.

Observe that the implicit guard signaling is complete, but not necessarily minimal.

To put our signaling scheme to a test we can list the effects of the initiations and of the terminations of all sections on the truth of all initiation conditions in a 'system table'. We can then verify that every effect that can remove a block is linked to the appropriate (conditional) signaling action. We will however not elaborate this further here.

As a final step towards the implementation of our coordination scheme we can replace all expressions with gate variables by expressions with communals.

In the readers and writers' scheme considered here we introduce two communals for this purpose: a boolean *writeractive*, and an integer *nrreaders*.

The following correspondences hold:

$$nrreaders = I(AR) - T(AR)$$

$$writeractive \rightarrow I(AW) > T(AW)$$

$$\text{NOT } writeractive \rightarrow I(AW) = T(AW)$$

We thus arrive at the following refined scheme:

```
SYSTEM readers&writers1;
VAR    nrreaders: communal integer;
       writeractive: communal boolean;

SECTION AR;
    IC: [NOT writeractive];
    IE: [nrreaders := nrreaders + 1];
    SP: [continuefq(AR)];
    "read;"
    TE: [nrreaders := nrreaders - 1];
    TP: [nrreaders = 0 → continuefq(AW)];
END AR /

SECTION AW;
    IC: [nrreaders = 0 ∩ NOT writeractive];
    IE: [writeractive := true];
    "write;"
    TE: [writeractive := false];
    TP: [W(AR) > 0 → continuefq(AR) / continuefq(AW)];
END AW;

INITIAL: [nrreaders := 0; writeractive := false];
END readers&writers1;
```

At this point the refined scheme can be translated straightforwardly into a monitor structure, as will be illustrated below.

5.4. Translation into monitor procedures

We will now be more specific about the nature of the reading and writing actions that have to be coordinated.

Let us assume that both the readers and the writers repeatedly want to access a single data field in a shared memory (e.g. one specific page). The readers' task is to read the data via a standard procedure:

readinfo(VAR block: UNIV page)

and to display it on a console via standard procedure:

displayinfo (block: UNIV page; where: 1..R)

Every reader should have a private console. We will use the readers' identity to address the consoles.

The writers' task is to scan specific peripherals for new data via standard procedure:

scaninfo (where: 1..W; VAR block: UNIV page)

and to update the data field accordingly via standard procedure:

writeinfo (block: UNIV page)

The writer identity should determine the peripheral(s) that have to be scanned.

In the user processes we would like to use abstract calls of the type

readers&writers1. ar (block), and
readers&writers1. aw (block)

to execute section AR or AW, respectively. Observe that the section bodies can be standardized and hidden from the user-level.

In the following we will assume that the procedures scaninfo and displayinfo are non-critical.

In Concurrent Pascal we can then implement the scheme from number 5.3 in three layers. We need a monitor to implement the exclusion on the execution of section headings and tails, we need a class to standardize the ordering of headings, bodies and tails, and we need another class to implement the multiprocess queues with fifo ordering.

In that order a full translation yields the following structures.

All types are declared globally. R and W are constants.

```
TYPE r&w-guard = MONITOR;
VAR nrreaders: integer;
   writeractive: boolean;
   arq: ARRAY[1..R] OF queue;
   awq: ARRAY[1..W] OF queue;
   reader, writer: fifo;

PROCEDURE ENTRY ar-head;
BEGIN
   IF writeractive THEN delay(arq [reader.in]);
   nrreaders := nrreaders + 1;
   continue(arq [reader.out])
END;

PROCEDURE ENTRY ar-tail;
BEGIN
   nrreaders := nrreaders - 1;
   IF nrreaders = 0 THEN continue (awq [writer.out])
END;

PROCEDURE ENTRY aw-head;
BEGIN
   IF writeractive OR nrreaders > 0
   THEN delay (awq [writer.in]);
   writeractive := true
END;

PROCEDURE ENTRY aw-tail;
BEGIN
   writeractive:= false;
   IF reader.waiting THEN continue (arq [reader.out])
   ELSE continue (awq [writer.out])
END;

BEGIN nrreaders := 0; writeractive := false;
   INIT reader(R), writer(W);
END;
```

An instance of this monitor is declared and initialized globally:

```
VAR protector: r&w-guard;
INIT protector;
```

The monitor instance "protector" is passed to all user processes as a parameter. Within each user process we declare an instance of the type r&w, which uses the protector for the coordination:

```
TYPE r&w = CLASS (protect: r&w-guard);

PROCEDURE ENTRY ar(VAR whatsnew: UNIV page);
BEGIN
   protect.ar-head;
   readinfo (whatsnew);
   protect.ar-tail
END;

PROCEDURE ENTRY aw(thatsnew: UNIV page);
BEGIN
   protect.aw-head;
   writeinfo(thatsnew);
   protect.aw-tail
END;

BEGIN END;
```

The class fifo is declared as follows. Observe that the function-entry "out" returns the index of an empty queue when the queue length is zero.

```
TYPE fifo = CLASS (limit: integer);
VAR top, bottom, length: integer;

FUNCTION ENTRY in: integer;
BEGIN
   in := top;
   top := (top MOD limit) + 1;
   length := length + 1
END;

FUNCTION ENTRY out: integer;
BEGIN
   out := bottom;
   IF length > 0 THEN
   BEGIN
      bottom := (bottom MOD limit) + 1;
      length := length - 1
   END
END;

FUNCTION ENTRY waiting: boolean;
BEGIN waiting := (length > 0) END;

BEGIN top := 1;  bottom := 1; length := 0 END;
```

The user processes can be declared as follows:

```
TYPE readers = PROCESS (who:1..R; protector: r&w-guard);
VAR whatsnew: UNIV page;
   readers&writers1: r&w;
BEGIN
   INIT readers&writers1 (protector);
   CYCLE
      readers&writers1.ar(whatsnew);
      displayinfo(whatsnew, who);
```

```
    ....
  END
END;

TYPE writers = PROCESS (who:1..W; protector: r&w-guard);
VAR thatsnew: UNIV page;
  readers&writers1: r&w;

BEGIN
  INIT readers&writers1 (protector);
  CYCLE
    ....
    scaninfo(who, thatsnew);
    readers&writers1.aw(thatsnew)
  END
END;
```

Instances of user processes, finally, are declared and initialized globally as follows:

```
VAR readproc: ARRAY [1..R] OF readers;
  writeproc: ARRAY [1..W] OF writers;
  index: integer;
  protector: r&w-guard;
  ......
INIT protector;
FOR index := 1 to R DO
  INIT readproc[index](index, protector);
FOR index := 1 to W DO
  INIT writeproc[index](index, protector);
```

The crucial part of this system is of course the monitor. Fortunately, the monitor structure can be derived directly from the refined system, even mechanically. To demonstrate this a translator program for the refined schemes has been written in Sequential Pascal. It operates on a PDP 11/55 system, supported by the SOLO operating system.

The monitor procedures delivered by the translator for the examples discussed here are, as expected, in many ways equivalent to earlier published solutions, for instance those of Hoare [9]. The advantage of the design method introduced here is clearly that coordinations can be built and analyzed step by step. Furthermore, the descriptions can be analyzed before they are implemented, before much of the clarity of a solution is necessarily lost in implementation details, or sacrificed to efficiency.

6. Refined systems

In the following we summarize the refinements for the three systems discussed in number 4. Designing minimal and complete signaling schemes for these systems is a fairly straightforward task. This is, however, not always the case. To appreciate the subtleties of this design phase the reader is encouraged to design a signaling scheme for a monitor solution of the dining philosophers' problem [7]. This problem exemplifies the difficulties one encounters when signaling tasks have to be transferred from one clause in the refined scheme to another.

To illustrate the general approach, though, the following examples may prove helpful.

6.1. Refined bounded-buffer scheme

We introduce one extra communal variable named *length* to indicate the number of portions in the buffer.

The correspondence is:

$$length = T(P) - T(C)$$

Because we have only a single producer we know that when IC(P) is evaluated we must have

$$I(P) = T(P)$$

IC(P) can therefore be replaced by the equivalent condition:

$$IC(P) = [T(P) - T(C) < N]$$

which can now be expressed as

$$IC(P) = [length < N]$$

Similarly, because we have only a single consumer process, we can express IC(C) as:

$$IC(C) = [length > 0]$$

the remaining refinements are trivial:

```
SYSTEM bounded-buffer;
VAR top, bottom, length: communal integer;

SECTION P(VAR pr1: integer);
   IC: [length < N];
   IE: [pr1 := top];
   "produce in buffer(pr1);"
   TE: [top := (top MOD N) +1; length := length + 1];
   TP: [continuefq(C)];
END P /

SECTION C(VAR pr2: integer);
   IC: [length > 0 ];
   IE: [pr2 := bottom];
   "consume from buffer (pr2);"
   TE: [bottom := (bottom MOD N) + 1; length := length - 1];
   TP: [continuefq(P)];
END C;

INITIAL: [top := 1; bottom := 1; length := 0];
END bounded-buffer;
```

6.2. Refined scheduled-wait scheme

We use the implicit queue of section wait with priority scheduling. For this we have to explicit the queueing part.

```
SYSTEM scheduled-wait;
VAR now: communal integer;

SECTION wake-me (N: integer);
VAR alarmset: communal integer;
   IE: [alarmset := now + N];
   SECTION wait;
      IC: [ now ≥ alarmset];
      QP: [enterq(wait, alarmset)];
      SP: [resumefq (wait)];
      skip;
   END wait;
END wake-me /
```

```
SECTION tick;
   IE: [now := now + 1];
   SP: [resumefq (wait)];
   skip;
END tick;

INITIAL: [now := 0];
END scheduled-wait;
```

6.3. Refined second readers and writers' scheme

To our surprise it appears that the refined scheme only differs from the simple scheme derived in 5.3 in one single aspect. We only have to replace IC(AR) with:

$$IC(AR) = [\text{NOT } writeractive \cap W(AW) = 0]$$

Observe that the introduction of communal variable *pref* was needed to secure the observance of Hoare's requirement:

"all readers waiting at the end of a write should have priority over the next writer".

But then, we took care to observe precisely that requirement in the design of the signaling scheme in number 5.3 when we had to guarantee strong reader priority. Applying the same signaling strategies to the second readers and writers' scheme makes the occurrence of *pref* in IC(AW) redundant.

The introduction of *pref* in IC(AR) was needed to suppress the blocking effects of W(AW) when readers had to be restarted. However, the same effect can be obtained by suppressing the reevaluations of IC(AR) in the signaling of readers in TP(AW).

It follows that if we can delete the variable *pref* from all initiation conditions, we can delete it entirely, and omit IE(AR) and TE(AW) from the non-refined scheme (see number 4.3).

It turns out that the introduction of variable *pref* in the scheme of 4.3 served only to make explicit the observance of a signaling requirement on a high level of abstraction. In fact, the deletion of *pref* from IC(AR) is an optimization which leads to a violation of axiom A(1), as readers can now be restarted while IC(AR) is false due to W(AW) > 0. It may well be argued that such optimizations should not be applied in this design phase, but only after the translation to, for instance, monitor procedures. In this case, however, it is not difficult to prove that the dissident scheme is functionally equivalent to a (less optimal) scheme which does obey axiom A(1).

7. In conclusion

We have shown that, with the aid of a simple axiomatic model, one can design coordination schemes step by step, and that each step can be analyzed on its correctness with convincing informal arguments.

We have illustrated how the schemes derived with this method can be implemented with standard monitor procedures, as they are available in Concurrent Pascal. The applicability of the model is, however, not restricted to monitor-like implementations. When using a concept like the guarded region, supported by a multiprocessor system as described in [4], the optimizing phases can be skipped entirely and the coordination schemes can be implemented directly.

As a more general analysis tool the model has been applied to the study of coordination problems in a multiprocessor controlled real-time system (a telephone switching system) [10]. The first study indicates that the use of the model can greatly improve our understanding of the required coordinations in such systems.

It seems that our method could be formalized still further in line with, for instance, Keller's models for the formal verification of parallel programs [12]. There is, however, a danger that in doing so we lose the conceptual simplicity of the model, while adding only slightly to its analytical power.

The soundness and completeness of the axiom system have not been discussed in this paper, but

surely deserve attention. We would like the axiom system to be powerful enough to allow us to prove, for instance, whether arbitrary states of the coordinated system are reachable, and whether or not specific execution histories are feasible.

We have shown how one can design and analyze coordination schemes and signaling schemes in separate design phases. The *optimality* of a coordinated system is, unfortunately, not uniquely determined by the signaling scheme. It is influenced by many factors which the programmer traditionally prefers to ignore, e.g. the specific implementation of the scheduling policy on a multiplexed machine. These problems also deserve further consideration.

The body of an atomic section is, in the present model, defined as an uninterpreted block of code, for which the only restriction is that its execution has no effect on the enforcement of the coordinations (i.e. no access of any state variable). An extension of the model may be to allow for the enforcement of restrictions on the possible contents of these bodies. We may for instance, restrict the 'options' of the users to some small predefined set of procedures per section. We can also use the parameter-passing mechanism to enforce a protection. The user must then provide a 'key' to the guard which is validated in the section head and invalidated in the tail. This key should be read-only for users.

A more obvious extension of the design method introduced here would be to include a second refinement step in which the single 'global' monitor derived is split into a maximal set of non-interfering smaller monitors, which can be accessed concurrently.

The model introduced here can serve as a basis for these further studies.

Acknowledgement -

It is a pleasure to acknowledge the support I had from Professors J.L. De Kroes and W.L. Van der Poel in the early phases of this work. I'm also grateful to Per Brinch Hansen for giving me the opportunity to continue this research during my stay at USC. Doug McIlroy provided many useful comments, leading to important improvements in style and presentation. Susan Massotty deserves credit for her competent correction of my English. I thank Bob McDonnel for uncovering an error in an earlier version of this paper. Finally, I thank my students at USC for convincing me that the design method I taught them really worked.

REFERENCES

1. **Belpaire, G. (1975),** *On programming dependences between parallel processes.* Techn. report no. 244, Comp. Science Dept., University of Wisconsin, Madison, March 1975, 61 pgs.
2. **Belpaire, G. & Willmotte, J. P. (1973),** *A semantic approach to the theory of parallel processes.* Proc. Intern. Computing Symp., 1973, Davos (Sw.), ACM, pp 159-164.
3. **Bernstein, P. A. (1973),** *Description problems in the modeling of asynchronous computer systems.* Techn. report No. 48, Dept. Computer Science, University Toronto, 1973, 116 pgs.
4. **Brinch Hansen, P. (1978),** *Multiprocessor architectures for concurrent programs.* ACM 78 conference, Washington, D. C., Dec. 1978, 20 pgs.
5. **Campbell, R. H. & Habermann, A. N. (1974),** *The specification of process synchronization by path expressions.* Lecture notes in Computer Science, Vol. 16, Springer Verlag, Berlin, 1974, pp. 89-102.
6. **Courtois, P. J., Heymans, T., & Parnas, D.L. (1971),** *Concurrent control with readers and writers.* Comm. ACM, Vol. 14, No. 10, Oct. 1971, pp. 667-668.
7. **Dijkstra, E. W. (1965),** *Cooperating sequential processes.* Report University of Techn. Eindhoven (Netherlands), Math Dept., No. EWD 123, Sept. 1965. Also in: Programming Languages, F. Genuys (ed.), Academic Press, New York, 1968.
8. **Greif, I. (1975),** *Semantics of communicating parallel processes.* Diss., MIT, Cambridge, Mass., 1975, 161 pgs. See also: Comm. ACM. Vol. 20, No. 12, Dec. 1977, pp. 931-935, for an overview.
9. **Hoare, C. A. R. (1974),** *Monitors, an operating system structuring concept.* Comm. ACM, Vol. 17, No. 10, Oct. 1974, pp. 549-557. Errata in Vol. 18, No. 2, Febr. 1975.
10. **Holzmann, G. J. (1979),** *Coordination problems in multiprocessing systems.* Diss. University of Techn. Delft (Netherlands), June 1979, 315 pgs.
11. **Howard, J. H. (1976),** *Proving monitors.* Comm. ACM, Vol. 19, No. 5, May 1976, pp. 273-279.
12. **Keller, R. (1976),** *Formal verification of parallel programs.* Comm. ACM, Vol. 19, No. 7, July 1976, pp. 371-384.
13 **Owicki, S. & Gries, D. (1976),** *Verifying properties of parallel programs: an axiomatic approach.* Comm. ACM, Vol. 19, No. 5, May 1976, pp. 279-285.
14. **Petri, C. A. (1962),** *Kommunikation mit Automaten.* Diss., University Darmstadt (in German).
15. **Presser, L. (1975),** *Multiprogramming coordination.* Computing Surveys, Vol. 7, No. 1, March 1975, pp. 21-44.
16. **Reed, D.P. & Kanodia, R.K. (1979),** *Synchronization with eventcounts and sequencers.* Comm. ACM, Vol. 22, No. 2, Febr. 1979, pp. 115-123.
17. **Robinson, L. & Holt, R. C. (1973),** *Formal specifications for solutions to synchronization problems.* Stanford Research Institute Report, Computer Science Group, Menlo Park, Cal., Nov. 1973, 25 pgs.
18. **Slutz, D. R. (1968),** *The flow graph schemata model for parallel computation.* Diss., MIT, MAC-TR-51, Sept. 1968.
19. **Wodon, P. L. (1972),** *Still another tool for synchronizing cooperating processes.* Report Carnegie-Mellon University, Dept. of Computer Science, Pittsburgh, Pa., August 1972.

APPENDIX

D-operations

In D-operations the traditional link between the testing and the setting of a semaphore variable, which occurs in the test&set operation and in the semaphore P operation, is removed.

In the simplest case one can define the following three indivisible operations [2,19].

1. Passage operation

   Notation: "s:", where s is the name of a semaphore variable. This operation can only be completed if $s \geq 0$. If s is negative then the process which attempts to execute the passage is blocked. The process is only released when the value of s is positive again.

The precise way in which processes are being blocked and released is not specified. We will assume that the longest waiting process is released first. Observe that the execution of a semaphore passage operation does not affect the value of the semaphore variable.

2. Down operation

   Notation: "DOWN s", where s is the name of a semaphore variable. A process cannot be blocked while executing the down operation. The operation is indivisible. The effect is: s := s − 1.

3. Up operation

   Notation: "UP s", where s is the name of a semaphore variable. A process cannot be blocked while executing the up operation. This operation is also indivisible. The effect is: s := s + 1.

With these three operations one can make the following combinations:

s_1:..:s_n: DOWN $t_1,t_2,..,t_m$ UP $u_1,u_2,..,u_k$;

Where n, m, and k are positive integers. This operation as a whole is again indivisible, and can only be executed if the following condition is fulfilled:

$$\sum_{i=1}^{n} s_i \geq 0$$

The effect of the execution is that all t-variables are decremented, and all u-variables are incremented.

The working of the familiar P(s) and V(s) operations can be simulated with D-operations as follows:

$P(s) =$ s: DOWN s

$V(s) =$ UP s

It is quite remarkable, though, that in the original definition cited above, the combination of delay conditions in the composite passages is not a logical, but an arithmetical one.

Logical combinations are often much more convenient. We will define two alternative passage operations with logical combinations of delay conditions below. These boolean passages are called the **and** passage and the **or** passage.

4. **And** passage operation

   Notation: "$s_1 \cap s_2 \cap \cdots \cap s_n$:", where $s_1, \cdots, s_n$ are semaphore variables. The **and** passage can only be executed if the following condition is fulfilled:

   $$\min(s_1,s_2,\cdots s_n) \geq 0$$

   The evaluation of this condition is indivisible.

5. Or passage operation

   Notation: "$s_1 \cup s_2 \cup \cdots \cup s_n$:", where $s_1, \cdots, s_n$ are semaphore variables. The **or** passage can only be executed if:

$$\max (s_1, s_2, \cdots, s_n) \geq 0$$

The evaluation of this condition is also indivisible.

The **and** and **or** passages can be combined into composite passages, in which the desired logical combination of blocking conditions is indicated with square brackets. For instance, the composite boolean passage:

$$[s_1 \cap s_2] \cup s_3:$$

can only be passed if:

$$\max ((s_3, \min (s_2, s_1))) \geq 0$$

The boolean passages can be combined with the normal up and down operations.

As a last extension of the D-semaphore concept we define an indivisible operation which allows us to set a semaphore variable to a given value.

6. Set operation

   Notation: "SET (s,e)", where s is a semaphore variable and e is an expression which may contain references to the values of other semaphore variables. The operation is indivisible. The effect is simply: s := e. For more than one variable the notation is:

   $$\text{SET } (s_1, e_1; \cdots ; s_m, e_m)$$

   In a combined operation the leftmost variable is set first.

As an example of the application of these extended D-operations we give a direct implementation of the coordinated system for the second readers and writers' problem, discussed in number 4.3.

We introduce six semaphore variables, which represent the following abstractions from the scheme:

$$\begin{aligned} wr &= -W(AR), \\ ww &= -W(AW), \\ ar &= -(I(AR) - T(AR)), \\ aw &= -(I(AW) - T(AW)), \\ p1 &= -(pref + 1), \\ p2 &= pref. \end{aligned}$$

The translation yields the following algorithm:

```
COBEGIN
"readers:"          REPEAT
                              DOWN wr;
aw ∩ [ww ∪ p1]:               DOWN ar, p1 UP wr, p2;
                              "read;"
                              UP ar;
                    FOREVER;

"writers:"          REPEAT
                              DOWN ww;
aw ∩ ar ∩ p2:                 DOWN aw UP ww;
                              "write;"
                              UP aw SET (p1,wr-1;p2,-wr);
                    FOREVER
COEND.
```

There is a difference with the original scheme from 4.3: we cannot make the decrement of p1 conditional with the D-operations.