2

on correctness analysis

"In the temple of truth there are many niches (-)."
Johan Goudsblom (1960)

Nihilisme en Cultuur, pg. 111.
Arbeiderspers, Amsterdam, 1960,
(in Dutch).

Contents Chapter Two

1. Introduction pg. 65

2. Graphical Models 70

   1. Petri Nets 70
   2. Coordination Nets 77
   3. Slutz Graphs 83
   4. Bigraphs 88
   5. Coordination Graphs 96

3. Formal Specifications 101

   1. Robinson & Holt specifications 101
   2. Actor Model specifications 105
   3. Path Expressions 110

4. Models Based on Predicate Calculus 115

   1. Analysis methods for sequential programs 115
   2. Analysis methods for parallel programs 121

5. Conclusions 126

6. References Chapter Two 127

List of Abbreviations Used in Chapter Two 130

CHAPTER TWO

ON CORRECTNESS ANALYSIS

2.1. INTRODUCTION[1]

We will avoid using such general terms as 'correctness proving' or 'formal verification' in this chapter. Clearly, it can never be proven that some system or some solution is 'universally' correct. It can only be established that a system has specific properties. The number of criteria on which one can analyze the 'correctness' of a system is always finite, and precisely because of this finiteness the significance of a 'verification' is always restricted. It may well be that after one has successfully completed a correctness analysis cycle, one comes to realize the relevance of still other correctness criteria which may invalidate the 'assumed correctness' of the system considered.
Manna & Waldinger worded these notions as follows:

> "... we cannot speak of the correctness of a program in isolation, but only of its correctness with respect to some specifications. After all, even an incorrect program performs *some* computation correctly, but not the same computation that the programmer had in mind."
> (Manna & Waldinger '78, pg. 201).

In variation of the famous word of Dijkstra on program testing (see next page) we can, therefore, say: 'the correctness of a solution for specific criteria can sometimes be proven, but the correctness for all thinkable criteria can not'.
In a correctness analysis one is restricted to proving the obeyance of some properties and the absence of some other properties. The successful completion of such a correctness analysis does *not* prove that the considered system is correct. It only means that the analyzer was not able to prove that the solution was *in*correct.

The standard way to analyze the correctness of a program is 'testing' ('debugging'). One checks if a given program meets the requirements by trying it out on some typical and some boundary cases.
There are two major reasons why we must reject this method, especially for the analysis of coordination structures in multiprocessing systems.

(1) One simply cannot test a program, or a system of programs, on all its functions under all possible conditions. Especially, in a multiprocessing environment the number of trials required for an 'exhaustive test' would be insuperably large. One is thus forced to restrict oneself to only a finite number of selected test-cases, which makes it uncertain (to say the least) whether indeed all (relevant) errors will be discovered.
The problem outlined here is essentially equivalent to the well-known philosophical *induction* problem. We cannot avoid quoting Dijkstra's famous word at this point:

- - - - -

(1) For the List of Abbreviations used in this chapter, see page 130.

"... program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."
(Dijkstra '72, pg. 864).

(2) As seen from the user's point of view a multiprocessing system is partly an indeterministic machine in which the actual ordering of the computations is irreproducible[1]. Even when a test would reveal an error in, for instance, the process coordination patterns, it is highly unlikely that the same error can be reproduced. Brinch Hansen, therefore, concluded:

"Program testing is simply useless as a means of locating time-dependent errors."
(Brinch Hansen '77, preface).

Earlier the same notions had been expressed by (among others) Wirth:

"A multi-program - a program consisting of several concurrent activities - cannot be exhaustively 'debugged' by the usual heuristic methods."
(Wirth '69, pg. 489).

For these reasons we must consider program testing an unreliable tool in a correctness analysis.
The alternative, as worded by Dijkstra, is:

"The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness."
(Dijkstra '72, pg. 864).

The goal of a correctness analysis, in Dijkstra's words, then is "to raise the confidence level of a program" and not to prove 'universal' correctness.
Another key-word in Dijkstra's statement is the term 'convincing'. A large and complex proof can hardly raise the confidence level of a program, and that is also a great restriction on the practical value of many analysis tools.
The view that the adequacy of a program should be proven rather than tested is not at all new. **Elspas '72 (pg. 97)** reported that some preliminary thoughts in this direction were elaborated already by Goldstine & Von Neumann (see **Taub '61**). McCarthy presented a first tentative correctness proving method at an IFIP-congress as early as in 1962. He explained:

"Instead of debugging a program, one should prove that it meets its specifications, ..."

"A proof that is easy to find, not too long, and easy to check."
(McCarthy '63, pg. 22).

Most attention, until now, has been paid to the analysis of sequential programs. It lasted until 1966 before the first practical correctness analysis techniques became available. The existing methods today are mostly based on the early work of **Naur '66, Floyd '67** and **Hoare '69**.
In this study we are, however, not primarily concerned with the analysis of sequential programs, but instead with the analysis of multiprocessing systems. We are not so much interested in the question of proving that a given piece of code will perform a specific computation within finite time, when executed on a computer. We are interested here in the question of proving the *integrity* of the interaction patterns in a set of concurrent processes. We want to study what methods are available to prove that a certain type of process coordination

---

(1) The indeterminism is caused here by external factors (e.g. user-behavior), not necessarily by intrinsic randomness.

is *consistent* with the requirements. For instance, we would like to be able to prove absence of deadlocks and starvations, correct exclusions on specific pieces of code, and obeyance of complex precedence rules. A correctness analysis of this type is in a way a *prerequisite* to the analysis of the more specific properties of the constituent processes.
We can thus divide the task of analyzing a system of (interacting) concurrent processes in two phases:

(1) In the first phase it is analyzed whether all explicit coordination requirements are obeyed (exclusions, high level precedence rules, etc.).
It is verified that there are no undesirable side-effects of the coordination (deadlocks, starvations, unreachability of relevant system states).

(2) In the second phase it can then be verified whether the solution is also functionally correct. Using the results of the analysis in the first phase one can try to prove that the system as a whole will realize the desired functions (e.g. computations).

In the first analysis phase we consider two types of requirements, which we can call, following **Lamport '77**, *safeness* and *liveness* properties.
Safeness properties relate to the absence of undesirable behavior (deadlock, starvation, violation of system invariants).
Liveness properties relate to the presence of desirable behavior (reachability of relevant states, obeyance of explicit coordination rules like exclusions).

Remark:

The distinction between liveness and safeness properties is not stringent. Observe that items like reachability (liveness) and unreachability (safeness) of system states are complementary.

In this chapter we consider only analysis tools for *logical* correctness. Other important items, like syntactical or dynamical correctness analysis, are not treated. (The interested reader is refered to, for instance, **Brinch Hansen '73 (pg. 226-234)**, **Horning '73**, and to the periodical *Update*, **August '77, Vol. 2, no. 8, pg. 6 and 7.**)

We will study the adequacy of a number of analysis tools for process coordination problems. The tools are defined as formal models. To analyze a set of concurrent processes one can represent their relevant aspects in the 'language' of such a model and attempt to prove certain properties within that model.
The model formalizes only a small set of relevant aspects of the system considered, the other aspects remaining 'uninterpreted'. As a result of these simplifications it can be much easier to prove the obeyance (or absence) of specific properties in the model than to prove it directly. The power of a model lies in the simplification.

The equivalences (or correspondences) between the actual system and the partial representation of that system in a formal model should guarantee that the properties that can be proven within the model hold for the system being modelled as well.

The models we will consider here are divided into three broad classes:

- graphical representations
- formal specifications, and
- models based on first order predicate calculus.

First, we shall study models based on graphical representations of coordinated systems: Petri Nets **(Petri '62)**, Coordination Nets **(Patil '70)**, Slutz Graphs

(Slutz '68), Bigraphs **(Gostelow '71, '72; Cerf '72)**, and Coordination Graphs **(Belpaire & Wilmotte '73; Belpaire '75)**.
The models usually leave the specific computations performed by the constituent processes uninterpreted, and focus on their interaction links.

Next, we shall study more abstract descriptive (and prescriptive) models in which one can formalize coordination structures in a high level language: Robinson & Holt specifications **(Robinson & Holt '73; Robinson '75)**, Actor Model specifications **(Greif '75; Goodman '76; Greif '77)**, and Path Expressions **(Campbell & Habermann '74; Habermann '75; Flon & Habermann '76; Lauer & Campbell '75)**.

Finally, we shall discuss models based on formalizations in first order predicate calculus. The models are heavily based on verification techniques for sequential programs, notably the inductive assertion method developed by **Floyd '67** and **Hoare '69**. For this reason we give a brief outline of the origination and development of these techniques, before we consider the extensions for the multiprocessing cases. **(Hoare '74; Howard '76; Owicki '75, '76)**.

A profound study of all aspects of the models considered here is beyond the scope of the present discussion. We will concentrate on just one of the aspects which is the adequacy of the models, to represent coordination structures and to aid in the analysis thereof. If we criticize the models on these aspects then, we clearly do not mean to question the adequacy of the models as a whole[1]

The criteria which play a central role in the discussion are the following:

Conceptual simplicity

There must be a clear relation between the system being modelled and the model itself. The model must give a conceptually clear representation of the problem being solved. One should be able to distinguish in the model between computations and coordination, and one should be able to concentrate on the latter aspect. One should be able to study coordination structures effectively on varying levels of abstraction.

Flexibility

One must be able to represent a wide variety of coordination structures in the model. The model must be an aid in both analysis and design of coordination patterns.

Verifiability

It must (at least) be easier to analyze a system within the model than to analyze it directly. One must be able to analyze the modelled system on the obeyance of safeness and liveness properties (see above). Ideally the analysis of these properties must be trivially amenable to automated verifications.

Summarizing: the model should have sufficient analytical power and sufficient descriptive clarity. There must be clear transformation rules to construct a model for a system and vice versa.
Observe that the requirements may be conflicting, for instance, analytical power and descriptive clarity (cf. chapter 1, section 1.3.2.1).
Criteria similar to those outlined here, for the evaluation of system models have been described by **Bernstein '73**, and **Boute '78**.

- - - - -

(1) Petri-Nets, for instance, can be of great value in the design of switching circuits. Those aspects are, however, not relevant to the study performed here.

Our aim is *not* to give an abstract comparative overview of the modelling power of existing models. Such an overview can be found in **Agerwala '75**. Agerwala showed, among other things, that Bigraphs and Slutz Graphs are equivalent to ordinary (non-extended) Petri Nets qua modelling power *as such*, and that Coordination Nets are equivalent to an extended type of Petri Nets (i.e. Petri Nets in which 'negation arcs' are allowed, as will be discussed in section 2.2.1 under **Observations**).
Clearly, the conceptual simplicity and the feasibility of informal correctness arguments can be quite different for descriptions which are *functionally* completely equivalent.

2.2. GRAPHICAL MODELS

Bernstein reported that:

> "... there are at least twenty-five distinct models for parallel computation documented in the literature."
> **(Bernstein '73, pg. 4).**

The larger part of these models are formalized *token-nets* or *flow-graphs*. These models are mostly variants of respectively, the Petri Net model and the flow graph model of **Rodriguez '67** (not treated here).
For each of the models discussed here we shall give an informal description, study some applications, and conclude with an evaluation.

2.2.1. Petri Nets (Petri '62; cf. Agerwala '75)

Description
A Petri Net is a collection of *places*, *transitions* and directed *edges*. Every edge connects a place to a transition or vice versa. Places are graphically represented by circles, transitions by bars, and edges by directed arcs.
Informally, a place corresponds to a condition, and a transition corresponds to an event. The input places of a transition T (those places which are directly connected to T via an edge pointing to T) correspond to the conditions that must be fulfilled before the event corresponding to T can occur. The output places of a transition T then correspond to the effect of the occurrence of the 'event' T on the conditions represented by these places.
Each place which corresponds to a fulfilled condition is marked with at least one *token* (sometimes called a *stone*). The occurrence of an event is represented in the Petri Net as the *firing* of a transition. A transition is enabled to fire if there is at least one token in each input place. The effect of a firing is that one token is added to the markings of all output places of the fired transition, and that simultaneously one token is removed from the markings of all the input places of that transition.
Two transitions are said to be *conflicting* if they share at least one input place. If the shared input place contains precisely one token, both transitions will be enabled to fire; but only one of the two can actually fire. The firing of any one of the two will disable the firing of the conflicting transition.
By definition the firing of conflicting transitions is *mutually exclusive*.
By assigning zero or more tokens to each place in a Petri Net we obtain a *marking* of this net. Each firing creates a new marking. A series of firings is called an *execution sequence*.
If all possible execution sequences of a given net, for a certain initial marking M, are infinite, the marking M is said to be *live*.
If in a certain marking no transition is enabled to fire, the net is said to *hang up*.
Finally, a marking M is called *safe* if no execution sequence starting with M can lead to a marking in which any place has more than one token in it.

A number of theorems and properties has been proven for Petri Nets, but mostly only for simplified types. Two examples of such simplifications are:

- Petri Nets in which precisely one edge is directed to and from each place. Such nets are called *marked graphs*.
- Petri Nets in which all transitions have at most one input place and one output place. Such nets are called *transition diagrams*.

Note that in a marked graph there can be no conflicting transitions, and in a transition diagram there is no clear representation of composite (delay) conditions.

Application

We can attempt to discover inconsistencies in a coordination structure by modelling it in a Petri Net (deadlock, starvation, unreachability of relevant system states, etc.).

Two or more conflicting transitions can model a mutual exclusion relation. In figure 2.1 transitions T_1 and T_3 conflict on place P_1, as P_1 contains only one token. As a result the two sequences T_1; T_2 and T_3; T_4 are mutually exclusive.

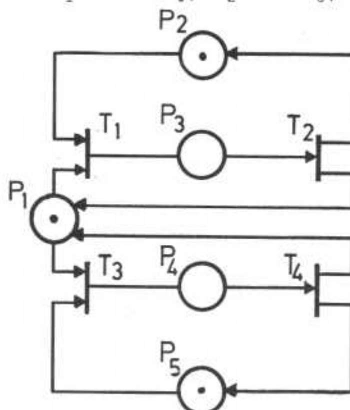


Figure 2.1.
Mutual Exclusion
(Petri Net)

Figure 2.2 models a potential deadlock in two concurrent processes. Initially both T_1 and T_3 are enabled. The sequence T_1; T_2 will consume a token first from place A and then from place B. The sequence T_3; T_4 is just the reverse. Each of the sequences T_1; T_4 and T_4; T_1 will lead to a hang up, which corresponds to a deadlock.

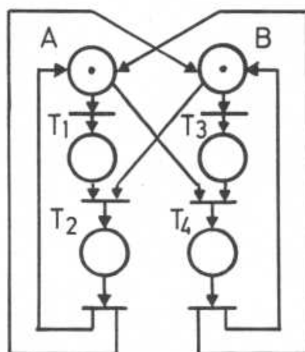


Figure 2.2.
Deadlock
(Petri Net)

Figure 2.3, finally, models a potential starvation loop (sometimes called a *livelock*, **(Keller '76; Ashcroft '75)**. Let $\oplus$ denote addition modulo 4, then every execution sequence

$$T_i;\ \left(T_{i\oplus 2};\ T_i';\ T_i;\ T_{i\oplus 2}'\right)^{\star}$$

will prevent the transitions $T_{i\oplus 1}$ and $T_{i\oplus 3}$ from ever being enabled. The symbol i is either 0, 1, 2 or 3.

The sequence placed between parentheses and superscripted by an asterix can be repeated indefinitely.

The processes represented by the sequences $T_{i\oplus 1}$; $T_{i\oplus 1}'$ and $T_{i\oplus 3}$; $T_{i\oplus 3}'$ are then starved, or 'live-locked'.

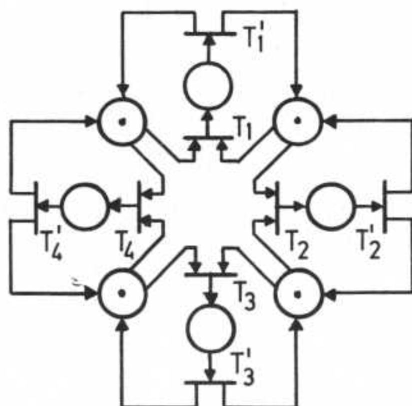


Figure 2.3.
Starvation
(Petri Net)

None of the three examples given can be represented transparently with merely marked graphs or transition diagrams.

Observations

(1) A token symbolizes not only the 'fulfillment of a condition', as we have described it above, but on another level of abstraction, also:

a. a *control point* in a program, indicating up to what point the execution of a sequence of operations has proceeded, and
b. a *privilege* to proceed beyond a certain point.

The first abstraction corresponds to the notion of ordering or sequencing of processes, the second abstraction corresponds to exclusion. Both symbolize the enforcement of partial orderings on the set of possible execution sequences (transition firings).

(2) The tokens in a Petri Net can only model control-flow points convincingly if the Petri Net structures allow us to model the three general control-flow structures straightforwardly (concatenation, selection, and iteration). A concatenation is easily modelled as a sequence of places and transitions. Selections and iterations, however, present more problems.
In the partial net of figure 2.4 we have attempted to model a selection. The 'control-flow token' travels either from place P0 via P1 to P3, or from P0 via P2 to P3. The problem is in the modelling of the working of the selection predicate. We can model the condition represented by the predicate in a place P4. The effects of place P4 on transitions T_1 and T_2 should, however, be complementary. We should, therefore, assign a 'complementary place' to P4, which is P4'. Whenever P4 contains a token, P4' must be empty, and vice versa.
A similar solution can be found for the modelling problems of iteration conditions. The resulting Petri Net structures can, however, hardly be called transparent. Note also that the transitions T_1 and T_2 in figure 2.4 are, according to the definitions, conflicting, even though they can never be enabled simultaneously.

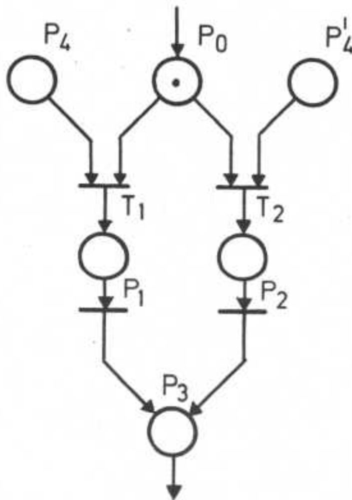


Figure 2.4.
Selection Structure
(Petri Net)

All this makes it rather difficult to distinguish between computation and coordination in a Petri Net.
The solution Agerwala gives to this modelling problem is to leave the flow-of-control in selection and iteration structures *uninterpreted*. Agerwala first extends the Petri Net model with definitions for, among others, exclusive-or (EOR) output logic of transition firings. This EOR logic can then be used to model an indeterministic flow-of-control in selection structures and iterations.

(3) We now consider the relations between tokens and execution *privileges*. There is a pleasant correspondence between the effect of the firing of a transition in a Petri Net, and the execution of parallel semaphore operations (as discussed in section 1.3.3.2). The values of the semaphore variables are represented by the number of tokens in the input places of a transition. The number of tokens in a place determines the maximum number of firings of the transitions which are in conflict over that place.
The input logic of a transition in the ordinary Petri Net model is by definition always an AND logic (i.e. a logical AND combination of firing conditions), just as with the conventional (non-extended) parallel P-operations.

(4) If we consider the places in a Petri Net as representations of general delay (or progress) conditions, and transitions as means to enforce delays, we must observe that one is rather restricted in the way in which one can combine, evaluate, and enforce such conditions. To compose firing conditions for the transitions one can indeed make logical AND combinations, but not much more than that. One can model a logical OR combination of conditions only indirectly (see e.g. **Agerwala '75**). Not only is the input logic of the transitions restricted to AND logic, but the output logic as well. As we have seen above (point 2) this makes the modelling of a flow-of-control unnecessarily difficult.

(5) The effect of the firing of a transition is also rather rigidly defined, especially if we consider the places as representations of conditions. It is not readily possible to increase or decrease the number of tokens in a place by any number other than precisely one per firing. (One can model such effects only by increasing the number of arcs, or by including input places also as output places of a certain transition.)

(6) Another complicating factor for the modelling of delay conditions is that the logical *not*-operation (negation) is also unavailable. With such a not-operation one would be able to fire a transition if a place contains no tokens, and to block the firing when it does contain tokens.

Remark:

The not-operation can be modelled in the Petri Net if the maximum number of tokens that can be assigned to a place in an execution sequence is known *a priori*. One can then assign a 'complementary place' to the place considered, initialized to this maximum *minus* the initial number of tokens in the place considered. The not-operation can then be constructed in the form of a 'maximum-test' on the complementary place. If one can remove the maximum number of tokens from the complementary place (and return them immediately in order to restore the correspondence between the number of tokens in the original place and the complementary place) then one knows that the original place must be empty. This maximum-test thus has the same effect as the execution of a not-operation (which is a 'zero-test') on the original place.

Agerwala suggested an extension of the Petri Net model with a special type of edge with models the not-operation. The special edge is indicated by a bar as shown in figure 2.5 (Agerwala used a slash.)

Figure 2.5. Not-Operation

Agerwala proved in his thesis that this extension effectively increases the modelling power of the Petri Net (unlike the extensions which allow for EOR input and/or output logic, etc.) **(Agerwala '75)**. The extension creates an interesting problem. In the net of figure 2.6 the transitions T_1 and T_2 are conflicting even though they do not share any input place. This forces us to rephrase the definition of a *conflict*:

> We call two or more transitions *conflicting* if the firing of any one of them can disable the firing of all others. Conflicting transitions may only be fired one at a time.

Note that with this new definition we obtain an exclusion on the firing of transitions T_1' and T_2' in figure 2.6, even though they do not conflict.

(7) In figure 2.7 we give a Petri Net representation of the following solution to the third version of the readers & writers' problem, discussed in chapter 1:

```
readers:                                      writers:
                 down WR;                                          down WW;
(WW ∪ RP) ∩ AW:  down AR up WR;          (WR ∪ WP) ∩ AR ∩ AW:     down AW up WW;
                 read;                                             write;
                 up AR;                                            up AW;
                 set (RP,0;WP,1);                                  set (RP,1;WP,0);
                 ...                                               ...
```

To improve upon the clarity of the net we have allowed for some notational simplifications:

- If a place is both input- and output-place of the same transition, the net effect of the firing of that transition on the number of tokens in this place is null. The transition can only fire if the place contains at least one token. We represent this by a special arc called an 'affirmation arc' as shown in figure 2.8.
- A place which is in the input set of more than one transition is connected to those transitions via a so-called 'multi-tailed arc' (see figure 2.7).
- A place which is in the output set of more than one transition is connected to those transitions via a so-called 'multi-headed arc' (fig. 2.7).

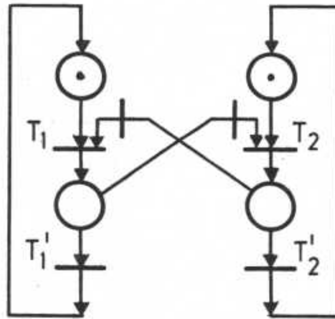


Figure 2.6.
Exclusion with Not-Operations
(Petri Net)

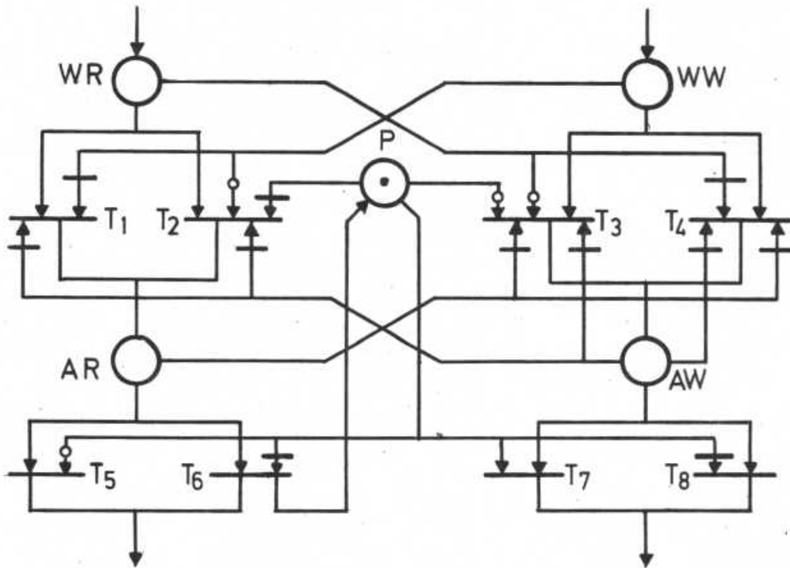


Figure 2.7.
Third Readers & Writers' Problem
(Petri Net)

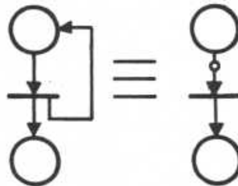


Figure 2.8.
Affirmation Arc

We have modelled the delay conditions for active readers and active writers as an OR combination of two AND compositions. The net structure unintentionally resembles that of the selection construct discussed earlier (see figure 2.4). Again we note that coordination aspects are hard to distinguish from control-flow aspects.
The setting and resetting of precedence condition P (representing WP and RP) is rather cumbersome. Observe that all ordering and exclusion relations are represented in the net; also those from which one would like to abstract.

Remark:

One can attempt to simplify a Petri Net somewhat by annotating the places with the conditions they represent, annotating the transitions with the effects they have (or with the operations which they represent), and deleting arcs which have no relevance to the problem studied. Such simplifications are, however, more palliatives than improvements to the Petri Net model.

(8) There are no standard procedures available for the analysis of a Petri Net on such items as deadlock and starvations. We may, however, think of the following procedure.
For the net of figure 2.7 we analyze whether reader processes can be blocked forever:

If readers are to be blocked then either of the two following partial markings of the net in figure 2.7 must be persistent for at least one possible future execution sequence[1]:

(a) the marking for $WW > 0 \cap P = 1 \cap WR > 0$,
(b) the marking for $AW > 0 \cap WR > 0$.

A third possibility is that the markings (a) and (b) can be reached alternately, and in overlap, for at least one possible future execution sequence. We consider each of these 3 cases below.

(a) With marking (a) transition T_3 is enabled to fire, *i*rrespective *o*f the *m*arkings of the other places in the net (we abbreviate this as: *iom*), provided that $AR = 0$.
Suppose, that $AR \neq 0$. It follows that $AR > 0$, which combined with the fact that $P = 1$ leads to the conclusion that T_6 is enabled (*iom*).
Our assumption that readers are starved implies that AR cannot be increased. The firing of T_6 will decrease it repeatedly until $AR = 0$. Within a finite number of firings AR must have reached the value zero, while P remains 1.
Returning to transition T_3, considered above, we conclude that T_3 will be enabled within finite time (*iom*). When T_3 fires, WW is decreased, and AW is increased, which makes $AW > 0$. But then, as $P = 1$, transition T_7 is enabled, and when T_7 fires it makes $P = 0$, which removes marking (a).
Conclusion: marking (a) is not persistent for any future execution sequence.

(b) Depending on the value of P, either T_7 or T_8 must be enabled. Transitions T_3 and T_4 are only enabled when $AW = 0$, so the value of AW cannot increase still further. Within a finite number of firings AW must have reached the value zero, which removes marking (b).
Conclusion: marking (b) is not persistent for any future execution sequence.

- - - - -

(1) The names of the places will be used to represent the number of tokens in them.

The third possibility is that markings (a) and (b) are reached alternately and overlapping. It is, however, easy to show that marking (b) must always lead to a marking in which AW = 0 AND P = 0 within finite time. In that state readers are not blocked (transition T_2 is enabled).

(9) The complexity of a Petri Net rises rapidly with the size of the problem being modelled. (As noted by, for instance, **Patil '70, (pg. 11)**, and **Bernstein '73.**)
To draw a clear net for interaction problems which involve more than two or three processes is almost impossible. We have further noted that there is no clear representation of general, composite delay-conditions in the (non-extended) Petri Net. The Petri Net descriptions are not always clear representations of the coordination problems being modelled. Computation aspects are sometimes hard to distinguish from coordination aspects.
There are no methods to simplify complex Petri Nets to smaller, and equivalent ones. Finally, it has been shown that coordination problems exist which the non-extended Petri Nets cannot represent correctly. (By Kosaraju, as noted in **Agerwala '75 (pg. 33)**; cf. also **Chen '75.)**

2.2.2. Coordination Nets (Patil '70)

Description
The Coordination Net is a generalization of the Petri Net. One of the differences is that the firing of a transition is divided into the following two steps:

(1) the claiming of a token from each input place of the transition considered,
(2) the removal of the claimed tokens, and the addition of one token to each output place.

Patil introduced the concept of the *constraint set*. If P is the set of places in a given Coordination Net, we can define the constraint set Ct as a subset of the *powerset* of P. Each member of the constraint set specifies a set of places which may not contain tokens simultaneously. Each entry of the constraint set is called a *constraint*. A constraint corresponds to a set of mutually irreconcilable conditions. Input places and output places of a certain transition may not be part of the same constraint.

A transition is called *active* if it has claimed tokens from each of its input places, but has not yet removed these.

A transition can only fire if the following two conditions are fulfilled:

(1) each of the input places contains at least one non-claimed token, and
(2) the marking which will result if the transition fires does not conflict with any of the constraints.

The execution of the two steps involved in the firing of a transition is indivisible, except for the so-called *output transitions*. In Patil's words:

> "... the initiation of an output transition indicates to the external world that it should proceed with a certain event associated with that transition. An output transition terminates only after the associated event has occurred."
> **(Patil '70, pg. 29).**

The constraint set can effectively simplify a complex Petri Net.
The nets of figures 2.1 and 2.6, for instance, can be represented by the Coordination Net of figure 2.9, with constraint-set: $Ct = \{ \{a,b\} \}$.

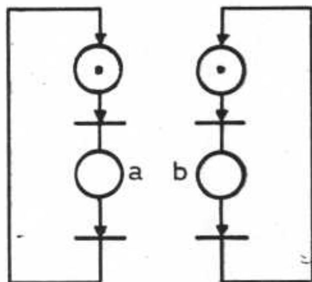


Figure 2.9.
Mutual Exclusion
Coordination Net
$Ct = \{ \{a,b\} \}$

If a constraint α is to be obeyed, clearly, all subsets ψ of the powerset of P with $\alpha \subseteq \psi \subseteq P$ are also constraints. We can then define a *reduced constraint set*, $R(Ct)$ as the smallest subset of Ct equivalent to it in the above sense. The *domain* of $R(Ct)$ is defined as the set of all places which occur in the entries of the reduced constraint set.

If a_i represents a place in set P and $R_{a_i}(Ct)$ the subset of the reduced constraint set which contains only those constraints in which a_i occurs, then the *influence set* of place a_i, symbolized as $I(a_i)$, is defined as the set which one obtains by deleting place a_i from each member of $R_{a_i}(Ct)$. (**Patil '70** uses slightly different terms.)
Two places a_i and a_j have *equivalent influence* if $I(a_i) = I(a_j)$.

A transition is called *homogenous* if its output places have equivalent influence. A Coordination Net is called homogenous if all transitions are homogenous.

The set of places P can be partitioned into equivalence classes of homogenous places. $EC(P)$ indicates the set of equivalence classes of P. The set $EC(Ct)$, finally, is obtained by replacing each place in each entry of the reduced constraint set by the (identification of the) equivalence class to which it belongs.

Example:

If: $P = \{P_1,\ldots,P_8\}$,

and: $Ct = \{ \{P_1,P_5\}, \{P_1,P_2,P_3\}, \{P_5,P_4\}, \{P_1,P_6,P_3\}, \{P_4,P_5,P_8\} \}$,

then: $R(Ct) = \{ \{P_1,P_5\}, \{P_1,P_2,P_3\}, \{P_5,P_4\}, \{P_1,P_6,P_3\} \}$.

The domain of the reduced constraint set is:

$$\{P_1,P_2,P_3,P_4,P_5,P_6\}.$$

The influence set of P_1 and P_2 is, respectively:

$$I(P_1) = \{ \{P_5\}, \{P_2,P_3\}, \{P_3,P_6\} \},$$
$$I(P_2) = \{ \{P_1,P_3\} \}.$$

Observe that place P_6 and place P_2 have equivalent influence.

$$EC(P) = \{ \{P_2,P_6\}, \{P_1\}, \{P_4\}, \{P_3\}, \{P_5\}, \{P_6\}, \{P_7,P_8\} \}.$$

If we indicate the first equivalence class listed in $EC(P)$ by E_1 and the i-th by E_i then we can write:

$$EC(Ct) = \{ \{E_2,E_5\}, \{E_2,E_1,E_4\}, \{E_3,E_5\}, \{E_2,E_6,E_4\} \}.$$

Clearly, the legality of the firing of a homogenous transition can be established by considering the influence set of *either* of its input places. If all places in at least one of the subsets of this influence set contain one or more tokens, the transition may not be fired. The enforcement of the constraints can thus be easier in a homogenous net than in a non-homogenous one.

A non-homogenous transition can be transformed into a sequence of homogenous

transitions, by making use of equivalence classes of output places **(Patil '70, pg. 45)**. The transformation involves an extension of the set of places P, and an extension of the set of constrains Ct.

Patil used the term *conflict cluster* for a collection of mutually conflicting transitions. The firing of any single transition from a conflict cluster may disable the firing of the other transitions in this set.

Application

In figure 2.9 we have given a Coordination Net for a mutual exclusion problem (compare this with the Petri Nets of figures 2.1 and 2.5).

In figure 2.10 we give a Coordination Net for a deadlock problem. The constraint set is:

$$Ct = \{ \{P_1,P_4\}, \{P_2,P_3\} \}.$$

There are two conflict clusters: (T_1,T_5) and (T_2,T_4). (Compare figure 2.10 with figure 2.2.)

In figure 2.11 we give a Coordination Net for a starvation problem.

The constraint set is:

$$Ct = \{ \{P_1,P_2\}, \{P_2,P_3\}, \ldots, \{P_i, P_{i+1}\}, \ldots, \{P_n,P_1\} \}.$$

There are n + 1 conflict clusters: (T_1,T_2), ..., (T_n,T_1). (Compare figure 2.11 with figure 2.3.)

The Coordination Net which is equivalent to the Petri Net of figure 2.7 can be found by deleting the arc from place AR to transitions T_3 and T_4, and the arc from place AW to transitions T_1 and T_2, and by defining the constraint set:

$$Ct = R(Ct) = \{ \{AR,AW\} \}.$$

This yields conflict cluster: (T_1,T_2,T_3,T_4).

Observe that although transitions T_1 and T_2, and similarly T_3 and T_4, are part of the same conflict cluster they can never be enabled simultaneously. Patil calls this a *pseudo-conflict*.

We further have:

$$I(AR) = \{ \{AW\} \},$$
$$I(AW) = \{ \{AR\} \},$$
$$I(WR) = I(WW) = I(P) = \{-\}.$$

There are three equivalence classes, namely:

$$E_1 = \{P,WR,WW\},$$
$$E_2 = \{AR\},$$
$$E_3 = \{AW\},$$

and thus we have:

$$EC(Ct) = \{ \{E_2,E_3\} \}.$$

Each of the transitions T_1, ..., T_8 is homogenous and thus the whole net is homogenous.

Observations

(1) In a Coordination Net one can formalize (some) exclusion relations in constraints. This formalization simplifies the graphical representation of the net and allows for a more rigid and transparent formal analysis of properties of the net. (Agerwala showed that the Coordination Nets are effectively more powerful than the non-extended Petri Nets. They can, however, formally be considered equivalent to the Petri Net model extended with the not-operation.)

(2) The major advantage of a Coordination Net over a Petri Net is that it allows

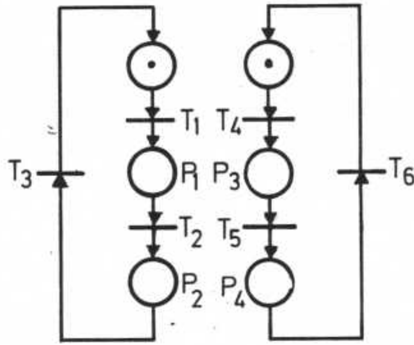


Figure 2.10.
Deadlock
Coordination Net
$Ct = \{ \{P_1, P_4\}, \{P_2, P_3\} \}$

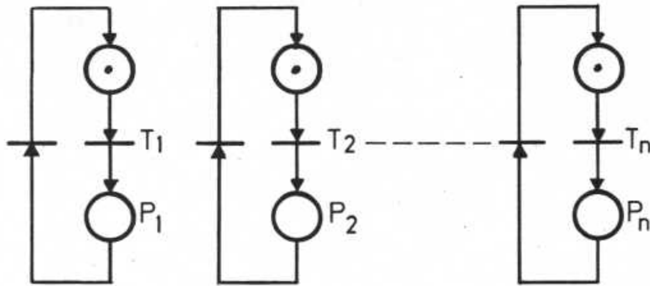


Figure 2.11.
Starvation
$Ct = \{ \{P_1, P_2\}, \ldots, \{P_i, P_{i+1}\}, \ldots, \{P_n, P_1\} \}$

us to distinguish (some) exclusion effects from ordering effects. Still, the tools provided are inherently restricted. With constraints we can formalize only one simple type of exclusion relation. The formalization cannot be applied to more complex relations, as occur, for instance, in the third version of the readers & writers' problem.

(3) In a constraint set we can formalize that certain partial markings of the net must be unreachable. That is a rather static approach. The power of the formalization tools could be increased if we were able to define a constraint set separately for each transition in the net. We then formalize merely that *certain* transitions may not lead into specific partial markings. Applying this to the solution of the third readers & writers' problem (fig. 2.7) we obtain a much greater simplification (see fig. 2.12).

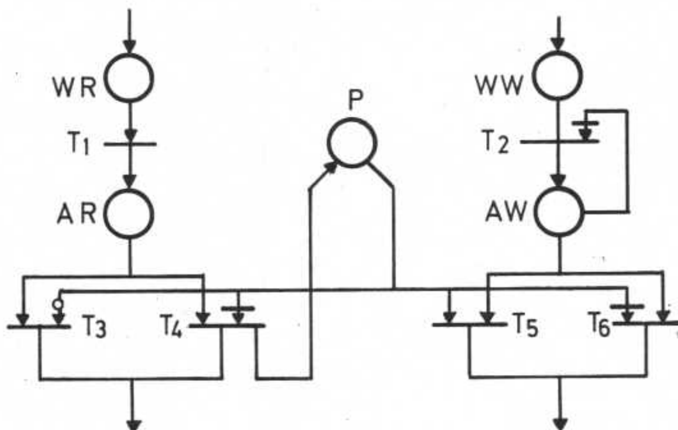


Figure 2.12.
Third Readers & Writers' Problem
Coordination Net

For simplicity we use the symbol $\bar{P}$ in the constraints to indicate that the place P may not be *empty* when the transition for which it holds fires.
If prefered, place P can also be replaced by two complementary places RP and WP, which can be used to model the effect of, respectively, P and $\bar{P}$ in the constraints. The net itself will, however, be more complicated in the latter case.

Let $Ct(T_x)$ indicate the set of constraints for transition T_x.
Let $R(Ct(T_x))$ indicate the corresponding reduced constraint set, then:

$$R(Ct(T_1)) = \{ \{WW, AR, P\}, \{AR, AW\} \}, \text{ and}$$
$$R(Ct(T_2)) = \{ \{WR, AW, \bar{P}\}, \{AR, AW\} \}.$$

The directed edge from place AW to transition T_2, in figure 2.12, cannot be removed, as one cannot specify in a constraint that place AW may never contain more than one token. For similar reasons we cannot formalize the setting and resetting of P in constraints either.
Comparing the net in figure 2.7 to that in figure 2.12 we note that the net has become simpler, but has also lost some of its informative value.

(4) Instead of restricting the set of markings into which the firing of a transition may lead, we can also restrict the set of markings in which a transition may fire. The difference seems rather small, but it has a large im-

pact on the modelling power of a Coordination Net.
Let *IM(T)* be the set of (partial) markings for the Coordination Net in which T occurs, in which transition T may not fire. *IM(T)* is called the set of illegal firing states of T. The net of figure 2.12 can now be simplified into the net of figure 2.13, in which:

$$IM(T_1) = \{ \{AW\}, \{WW,P\} \},$$
$$IM(T_2) = \{ \{AR\}, \{WR,\bar{P}\},\{AW\} \},$$
$$IM(T_3) = IM(T_5) = \{ \{P\} \}, \text{ and}$$
$$IM(T_4) = IM(T_6) = \{ \{\bar{P}\} \}.$$

The net shows clearly how the preference condition (expressed in P and $\bar{P}$) is exchanged between readers and writers. The net specifies the effects of a firing, while the sets of illegal firing states specify the conditions.

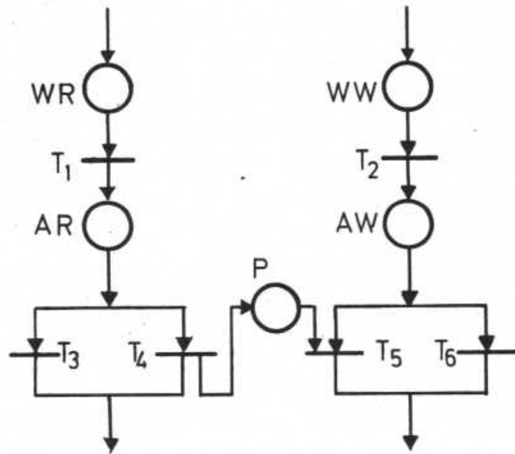


Figure 2.13.
Third Readers & Writers Problem
Simplified Coordination Net

(5) With the extensions suggested in point 3 we can make each net homogenous, without transformations. Observe that all output places of a non-homogenous transition T receive an extra token simultaneously, when T fires. If the firing of T is prohibited, because the addition of a token to the places of one of the equivalence classes in the output-set of T would violate a constraint, then *in effect* the same constraint holds for all output places of T. In other words: if a constraint holds for one place in the output set of T, it holds for all places in the output set of T.
We can, therefore, consider all output places of T as members of the same equivalence class, by combining all their constraints in one set. Some of the output places of T may be output places of still other transitions, so this combination is only correct with respect to T itself. For the other transitions we have to make other combinations of constraints. The entire Coordination Net can thus be made homogenous, by making each transition homogenous.

(6) There is a pleasant correspondence between the formalization of the illegal firing states and the working of (extended) dependence operations.
Compare the algorithm for the third readers & writers' problem, given in point 7 of section 2.2.1, with the Coordination Net in figure 2.13 and the constraints given in point 4 (this section). It is not difficult to deduct a transformation method from the correspondences between $IM(T_1)$ and $IM(T_2)$,

on the one side, and the semaphore passages in the readers & writers' algorithm, on the other side. If we apply such transformation rules to $IM(T_3)$, ..., $IM(T_6)$ as well, we obtain a solution which models the required coordination pattern with 2 concurrent processes for each single reader or writer process.
(In other words: 2 streams per functional process.) By introducing an indivisible SET operation we can reduce the number of 'streams' per functional process to one, and thus reconstruct the original algorithm.

(7) It follows from the above observations that Coordination Nets are better suited for the analysis of coordination problems than Petri Nets. With a few straightforward extensions we can establish pleasant correspondences between the working of the (extended) dependence operations and the firing of transitions in a Coordination Net. A Coordination Net is, however, not easier to find than an algorithm. The representations are conceptually not much clearer either.

2.2.3. **Slutz Graphs (Slutz '68)**

Description
Slutz Graphs (also called 'flow graph schemata' or 'computational schemata') can be considered as an extension of the parallel program schemata defined by **Karp & Miller '67, '69 (see Cerf '72, pg. 14 ; Bernstein '73, pg. 19).**

One uses two types of graphs:

- a *data flow graph* and
- a *control graph*

The processes, to be modelled in these graphs, are decomposed in *operations* which access *registers* (memory cells).

The data flow graph specifies the domain and range of each operation: a directed edge from operation A to cell N indicates that N is in the range of A, and reversely, an edge from N to A indicates that N is in the domain of A.
The operations, abstracted as *operators*, are represented by circles in the graphs. Memory cells are represented by squares.

The control graph specifies the sequence in which the operations are to be executed. Operations are again represented by circles. In the control graph the circles are connected via *control counters*. The function of these control counters is equivalent to that of the places in a Petri Net. Each control counter contains a nonnegative number which specifies a state. Edges branching out of a control counter can be annotated with a nonnegative number which indicates by what amount the control count must be decremented when the operation to which it leads is initiated. No counter may, however, be decremented to a negative number. If this restriction permits the initiation of an operation, that operation is said to be *defined*. Edges directed *to* a control counter can also be annotated with a nonnegative number, indicating by what amount the considered control count must be *in*cremented when the operation from which the edge stems terminates. In the latter case the annotation may also be a function of the contents of the cells which are in the domain of the terminated operation.

With these facilities we can model many types of output-logic in the Slutz Graphs (e.g. EOR-output logic, which is not possible in Petri Nets or Coordination Nets).

Example: (from Tsichritzis & Bernstein '74)

The computation of the function $f_1(f_2(x_1), f_3(x_2))$, in which x_1 and x_2 are variables and $f_1, \ldots, f_3$ are functions, can be represented as indicated in figure 2.14. For clarity we represent control counters by double squares, and memory cells by single squares.

r_1, r_2 and r_3 are cells (registers);
r_1 and r_2 are in the domain of f_1;
r_3 is in the range of f_1.

In the control graph the initial value of each control count is specified. The function f_4 is applied to the contents of register r_2.

An *interpretation* of a Slutz Graph consists of:

- the specification of the functions for each operator;
- the specification of the functions for each predicate used in the annotations of the edges branching *into* control counters (f_4 in the example), and
- the specification of the range of the contents of each cell. (optional).

An *execution sequence* is a series of initiations and terminations of defined operations in a Slutz Graph. The sequence is called *complete* if no operation in the graph is defined after it has been executed. (Note that completeness can imply deadlock.) With each execution sequence we can associate a *history* of values for each cell in the data flow graph.

Two operations *conflict* if the range and/or domain of one operation overlaps the range of the other. If two or more conflicting operations can be defined at

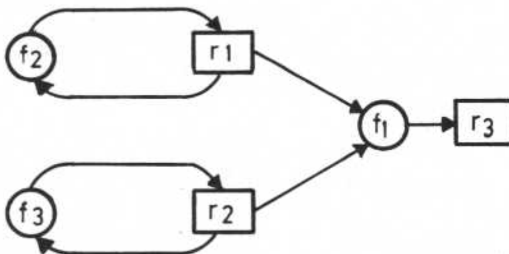


Figure 2.14a.
Data Flow Graph
$f_1(f_2(x_1), f_3(x_2))$

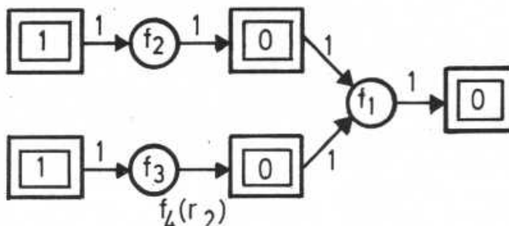


Figure 2.14b.
Control Graph
$f_1(f_2(x_1), f_3(x_2))$

the same time there is a *race condition*.
A schema is called *completely functional* if the history of all cells is the same for every possible execution sequence. Observe that the existence of a race condition can destroy this functionally.
If we call all cells that are in the range of functions in the schema, but are not in the domain of any function *output cells*, we can call a schema *output functional* if it is completely functional with respect to these output cells.

For the analysis of control graphs Slutz has suggested a 'vector addition method' (see also **Karp & Miller '69**). Each control counter in the schema is associated with a vector element. One can then describe the effect of the execution of each operation with vectors of increment and/or decrement values, which can be added to the state vectors of the control counters (see below).
One can then analyze the working of the graphs by manipulating these increment/decrement vectors.

Application
In figure 2.15 we give a Slutz Graph model for a mutual exclusion problem. (Compare with figures 2.1, 2.5 and 2.9.)
The control counters are identified by the symbols M, N, and P. We will use the symbol "t" (from transition) to indicate functions, from this point on.
We can define a control counter state-vector [M,N,P], with initial value [1,0,0]. For each operation in the control graph we can then define a state transition in the form of an addition vector:

for t_1: [-1,1,0],
for t_3: [-1,0,1],
for t_2: [1,-1,0],
for t_4: [1,0,-1].

There are two possible execution sequences: (t_1,t_2) and (t_3,t_4). By adding the corresponding vectors we find in both cases the result [0,0,0], which implies that for both sequences the terminal state equals the initial state of the control counters.

In figure 2.16 we give two control graphs, and an example of a data flow graph, which can lead to a deadlock. We define in figure 2.16a the state vector [M,N,P,Q], with initial state [1,1,0,0]. The addition vectors per operation are:

for t_1: [-1,0,1,0],
for t_2: [1,0,-1,0],
for t_3: [0,-1,0,1],
for t_4: [0,1,0,-1].

When relevant we can split the execution of an operation into an initial part and a terminal part. The addition vectors for operations t_2 and t_4 then become:

for t_2(init): [0,-1,-1,0],
for t_2(term): [1,1,0,0],
for t_4(init): [-1,0,0,-1],
for t_4(term): [1,1,0,0].

There are four possible execution sequences, of which two are 'complete':

(1) $t_1;t_2$,
(2) $t_3;t_4$,
(3) $t_1;t_3$(deadlock),
(4) $t_3;t_1$(deadlock).

Figure 2.16b shows how the two control counters, N and M, can be combined in a single counter.

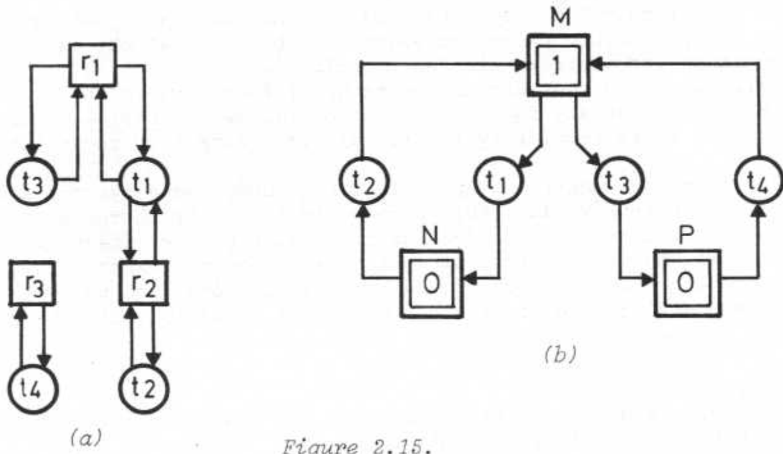
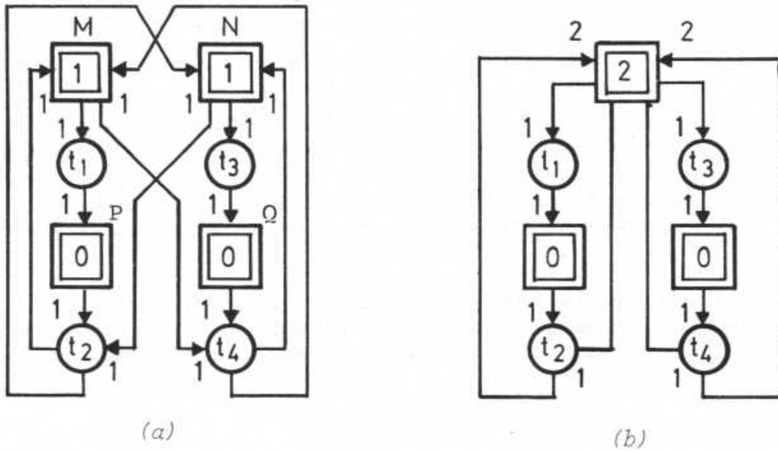


Figure 2.15.
Slutz Graph Mutual Exclusion Problem
Fig. 15a : Data Flow Graph,
Fig. 15b : Control Graph.



(c)

Figure 2.16.
Slutz Graph Deadlock Problem
Fig. 16a&b: Control Graphs,
Fig. 16c : Data Flow Graph.

(Compare figure 2.16 with figures 2.10 and 2.2.)
In figure 2.17 we give the Slutz Graph representation of a starvation problem. With every shared cell in the data flow graph, we associate a control counter in the control graph, initialized to 1. The control graph is more simple than the Petri Net from figure 2.3 and the Coordination Net of figure 2.11.

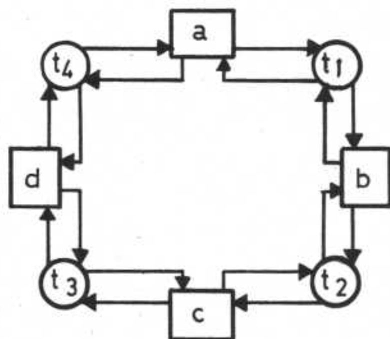


Figure 2.17a.
Starvation Problem
Data Flow Graph

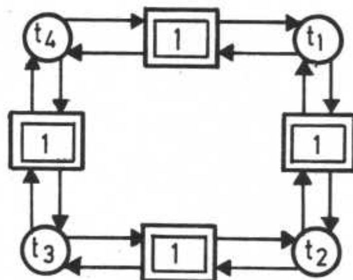


Figure 2.17b.
Starvation Problem
Control Graph

When we try to model the solution of the third readers & writers' problem, earlier represented in figures 2.7, 2.12 and 2.13, we run into serious troubles. Although, the structure of a control graph is nearly the same as the structure of a Petri Net or Coordination Net, there is an essential difference in the working. The execution of an operation in the Slutz Graph is no indivisible act, it consists of an initiation and a termination. This means that we can no longer silently assume that the value of a control count is unaffected by the execution of an operation if the termination of an operation increments the control count by the same number as the initiation of that operation has decremented it. The working of the 'affirmation arcs' discussed in section 2.2.1, point 7 is then different, and via these arcs one may unintentionally introduce undesirable exclusion effects in a solution. We can overcome this problem by introducing special indivisible 'null edges' in the control graphs which symbolize the working of an indivisible affirmation arc. The null edges are then the counterpart of the 'negation arcs' introduced earlier. The null edge represents a true initiation condition for an operation, if the value of the control counter from which it branches out of is positive. The initiation (or termination) of the corresponding operation does not affect the value of this control count via null edges. Still, we run into more trouble if we try to formalize the effect of the null edges and negation arcs in addition vectors. To circumvent these problems one would have to use graph transformations (see e.g. the remark in point 6, section 2.2.1) which complicate the graph and the application of the vector addition method considerably, and make them less suited for manual analysis.

Observations

(1) The facility in control graphs which allows us to annotate the arcs to and from control counters with numbers different from 1 reminds us of the extension from simple semaphore-operations to number semaphore-operations. We may expect that the descriptive power of a Slutz Graph is increased accordingly.

(2) The data flow graph provides us with additional, but often redundant, information about the nature of interactions. The data flow graph can be considered as part of the problem specification, and the control graph as part of its solution.

(3) At the point where we found the Petri Nets and Coordination Nets too restrictive in their choice for the modelling of one specific type of exclusion, we find the Slutz Graph too liberal. It is difficult to extend the Slutz Graph model (see above), and as it is, the Slutz Graph offers us hardly any tool for modelling coordination structures effectively. Observe that the facilities for the construction of exclusions in the model should match the facilities for the construction of exclusions in the (multiprocessing) system being modelled.

2.2.4. **Bigraphs (Gostelow '71, '72; Cerf '72)**

Description
A bigraph (bilogic directed graph) is a collection of vertices and directed arcs. The vertices represent computations; the arcs are used to enforce partial orderings on these computations. For each vertex in the bigraph one specifies two functions, called L and Q. The enforcement of the partial ordering is prescribed partly in the structure of the graph and partly in functions L and Q.

Each arc is named, and can be specified by an ordered pair of sets defining the initial and terminal vertices of this (directed) arc. Each arc can have zero or more initial and terminal vertices, provided it has at least one of either. Arcs with more than one initial vertex are called multi-tailed; arcs with more than one terminal vertex are called multi-headed. Arcs without an initial vertex are called *entry-arcs*; arcs without a terminal vertex are called *exit-arcs*. Each arc represents an enabling condition for the computation(s) represented by its terminal vertex (-ices).

Functions L and Q consist of two subfunctions each: L^- and L^+, and Q^- and Q^+. The L functions define the logical combination in which the 'enabling conditions' of a vertex must be, respectively, *evaluated* (input-arcs) or *set* (output arcs). Each arc can contain a nonnegative number of tokens. The function Q^- determines per vertex, and per input arc, how many tokens the arc should minimally contain to enable the initiation of the vertex.
The L^- function determines, per vertex, whether the ensemble of enabling conditions represented by the input arcs must be evaluated with AND logic or with EOR logic (combinations are not allowed).
The L^+ function similarly specifies whether the number of tokens on *all* or on *one* arbitrary output arc must be incremented when the vertex terminates.
The effect of the initiation of a vertex is that the number of tokens specified in the Q^- function is removed from all, or one, enabled input arc(s) (depending on the value of the L^- function).
The effect of the termination of a vertex is that the number of tokens specified in the Q^+ function is added to all, or one arbitrary, output arc(s), (depending on the value of the L^+ function).
The bigraph is conceptually processed by an abstract 'token machine', which attempts to initiate and terminate enabled vertices at random; one at a time. By specifying the contents of each computation in the bigraph we can give an *interpretation* of that graph. If we merely specify for each vertex the domain and range of the corresponding computation, we speak of a *partially interpreted bigraph* or a *GMC* (Graph Model of Computation, see **Cerf '72**).

Example: (from Cerf '72, pg. 5,6)

For a complete interpretation of the bigraph in figure 2.18 (in which + denotes EOR logic, and ★ denotes AND logic) we can specify each vertex as follows:

1: $x := c_1$; $e := c_2$; $t := c_3$;
2: $w := x^2$;
3: if $|t^2 - x| < e$ then goto 4 else goto 5;
4: $z := t + w$
5: $t := (t + x/t)/2$

The completely interpreted bigraph models the computation of the function $z := x^2 + \sqrt{x}$, where $\sqrt{x}$ is computed by Newton's method of iteration.
In figure 2.18 we have further:

$Q^- = Q^+ = 1$ for all vertices;
$L_1^+ = L_4^- = \star$ (AND logic), and
$L_3^- = L_3^+ = +$ (EOR logic).

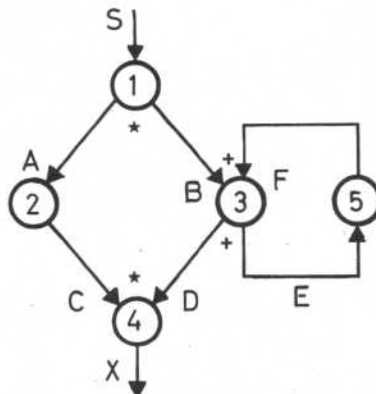


Figure 2.18.
Bigraph
$z := x^2 + \sqrt{x}$

We can describe the effect of the initiation and termination of vertices in *transformation expressions* (TE). For instance, the TE:

$$t^{\bar{1}}: S \rightarrow 1$$

describes that the initiation of vertex 1 (indicated as $\bar{1}$) causes a decrement of the number of tokens on arc S by one, and will mark vertex 1 *active* by placing a single token in it. Note that a vertex may be initiated more than once, and may contain more than just one 'active' token.
Similarly, the TE:

$$t^{\underline{1}}: 1 \rightarrow ABB$$

would describe that the termination of vertex 1 (indicated as $\underline{1}$) causes the number of tokens on arc A to be incremented by one, and for arc B by two. Simultaneously vertex 1 looses one of its 'active' tokens.

If the initiation or termination of a vertex can have more than one possible result (because of EOR input or output logic) we specify a different TE for each possibility, and distinguish between the TE's via subscripts:

$t_1^{\bar{3}}: 3 \rightarrow E$ and $t_2^{\bar{3}}: 3 \rightarrow D$.

The working of a bigraph can be completely specified in TE's. With the TE's we can construct a state diagram (called a CFG or *C*omputation *F*low *G*raph), of all states which the bigraph may reach while executing in the TM (token machine). The CFG specifies all possible execution sequences. As an example, the CFG of the bigraph in figure 2.18 is given in figure 2.19.

A *path* in the CFG can be defined as a sequence of TE's, such that for all pairs of successive TE's there must be a directed arc in the CFG pointing from the first to the second TE. Cerf and Gostelow use another notion of a path, which they relate to bigraphs instead of CFG's. The approach followed here lends itself better to the analysis of properties like deadlock and starvations in the CFG's.
In the CFG we specify in the vertices all active places and 'active arcs' in the bigraph (those containing tokens), the arcs are annotated with the event which they represent (either an initiation or a termination of a vertex in the bigraph).

A state A in the CFG is called a *minimum* if there is a path from A to every other state in that CFG.
If there is a path from A to every state in a subset β of the states in the CFG, then A is called a *minorant of* β. If A is itself a member of β then A can also be called a minimum of β.
Alternatively, if there is a path from every state in β to A, than A is called a *majorant of* β, and if A is an element of β then A can be called a *maximum of* β.
We may require that the terminal state in the CFG (the state in which we want the computation to stop, e.g. state 'X' in the CFG of figure 2.19) is a maximum of the CFG. If this is not so, the computation may stop in a deadlock state. The initial state of a computation ('S' in figure 2.19) should be a minimum of the subset of the CFG of desirable system states. For terminating computations the number of states in the CFG should further be finite.

Cerf and Gostelow have described procedures with which one can establish a property of a bigraph called *proper termination* (PT). According to their definitions, a bigraph with a *s*ingle *e*ntry and a *s*ingle e*x*it (SESX) is PT if for any interpretation:

- the processing of the bigraph can be performed with a finite number of tokens, and
- when the CFG reaches a terminal state then:
  - (a) there is precisely one token on the exit arc, and
  - (b) there are no tokens on any other arc in the graph.

The termination is 'proper' in the sense that it leaves the bigraph in a state with all arcs empty, except the exit arc.
The CFG may contain loops. PT does not imply that a bigraph will always terminate, not even if we would be able to prove that the terminal state is a minimum of the CFG. PT does imply the absence of deadlock, but not the absence of live-lock (starvations).
To determine whether a SESX bigraph is PT we can use a reduction procedure on the set of TE's. If the procedure reduces the set of TE's to only one TE (corresponding to the transition 'entry arc' → 'exit arc') the bigraph is PT (as was proven by Cerf and Gostelow).
The reduction procedure can be described as follows:
Let t be a TE, and let σ be an arbitrary symbol in a TE.
$n_L(\sigma,t)$ is defined as the number of instances of symbol σ in LHS(t), where LHS(t) denotes the left-hand side of expression t.

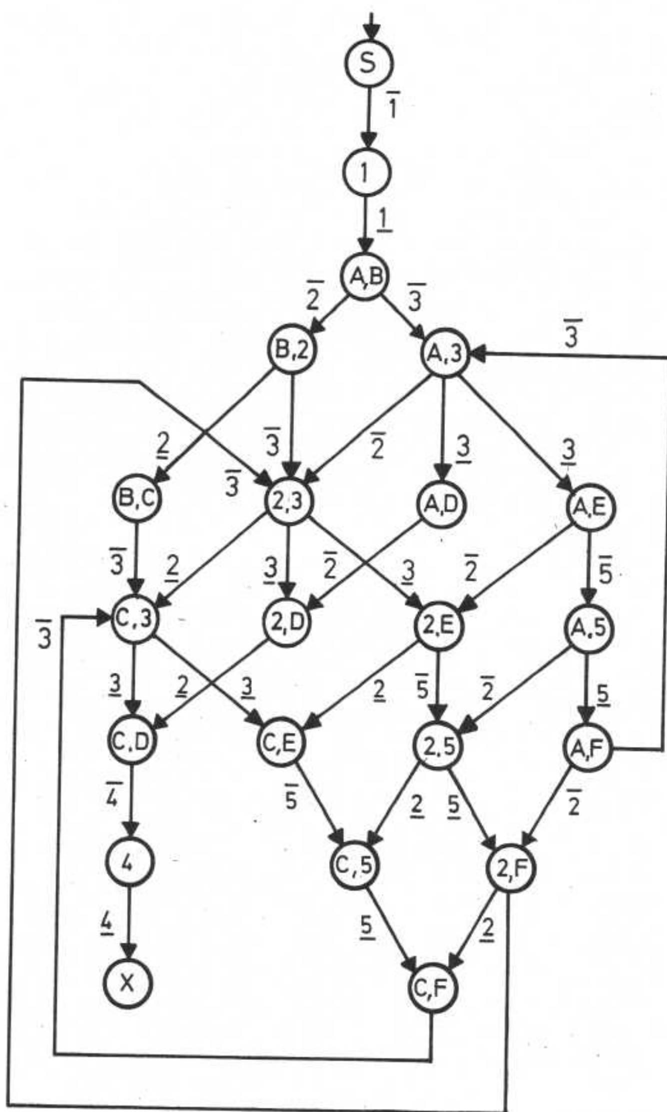


Figure 2.19.
Computation Flow Graph (CFG)
for the Bigraph in Fig. 2.18.

$n_R(\sigma,t)$ is defined as the number of instances of symbol σ in RHS(t), where RHS(t) denotes the right-hand side of expression t.
$r = t(d)$ denotes the result of the substitution of TE d in TE t.
Substitutions are defined as follows:

$$(\forall\sigma)\ \{(n_R(\sigma,r) := n_R(\sigma,t) - n_L(\sigma,d) + n_R(\sigma,d))\ \text{AND}\ \text{LHS}(r) = \text{LHS}(t)\}$$

provided that: $n_R(\sigma,r) \geqslant 0$.

Observe that the order in which symbols are listed on either side of a TE is irrelevant.
The reduction procedure consists of three steps:

(1) Delete all TE's with LHS(t) = RHS(t).
(2) Select a subset of TE's with identical left-hand sides. This subset may also contain only one TE. We will call the left-hand side of these TE's LHS(ϕ). Substitute every TE from this set consecutively in every TE t for which all symbols in LHS(ϕ) occur also in RHS(t), *provided* that the result of this sequence of substitutions is that none of the symbols in LHS(ϕ) occur in the new set of TE's which is obtained by *deleting* all TE's that were used in the substitution process, and *including* all TE's that were obtained via the substitutions. (All symbols in LHS(ϕ) must have been eliminated.)
(3) Repeat step 2 for the new set of TE's, until no more substitutions can be made. The considered graph is only PT if the completely reduced set of TE's contains precisely one TE.

Example:
For the bigraph in figure 2.18 we obtain the following set of TE's:

$t^{\bar{1}}$: S → 1
$t^{\underline{1}}$: 1 → A,B
$t^{\bar{2}}$: A → 2
$t^{\underline{2}}$: 2 → C
$t_1^{\bar{3}}$: B → 3
$t_2^{\bar{3}}$: F → 3
$t_1^{\underline{3}}$: 3 → D
$t_2^{\underline{3}}$: 3 → E
$t^{\bar{4}}$: C,D → 4
$t^{\underline{4}}$: 4 → X
$t^{\bar{5}}$: E → 5
$t^{\underline{5}}$: 5 → F

In the first reduction step we can eliminate symbol "1", by replacing it with "A,B". The first two TE's then converge to: S → A,B.
In the second step we eliminate "A", by combining the new TE with the third one, which yields: S → 2,B.
Consecutively we eliminate the symbols: "2", "B", "3", "F", "E", "5", "C", and "D", and finally also "4". The set of TE's is then reduced to one single TE, namely: S → X. As a further example of the reduction method: the elimination of symbol "3" in the initial set given above transforms the fifth, sixth, seventh, and eighth TE into:

B → D
B → E
F → D
F → E

(As there is more than one TE in which "3" occurs on the right-hand side, the *number* of TE's is not reduced by this specific elimination.)
The fact that the complete set of TE's can be reduced to one single TE proves that the bigraph of figure 2.18 is PT.

Gostelow '72 showed that the reduction procedure can be shortened by generating a smaller initial set of TE's from which all vertex symbols are deleted.

A CFG may contain loops, which implies that some states can be repeated indefinitely. To determine whether such repetitions are by necessity infinite or not, one introduced the concept of *repetition freedom* (RF).
(Gostelow '71; Cerf '72; Slutz '68). Cerf's definition of RF can be summarized as follows:

> A GMC is repetition free if for each *decision vertex* (vertex with EOR output logic and two or more outgoing arcs) which can be repeated, there is at least one other vertex which is necessarily always executed between two successive initiations of the decision vertex, and whose output data set overlaps the input data set of the decision vertex.

Observe that the property RF does not imply that infinite looping is impossible. It merely states that the GMC does not *necessarily* contain an infinite loop. As **Cerf '72** remarked **(pg. 66)** the above definition is quite restrictive:

> "Any GMC which contains a vertex which can be initiated more than once before it terminates cannot be RF."

We will return to this point in the discussion below.

Application

A bigraph representation of a mutual exclusion problem is given in figure 2.20. The multi-headed and multi-tailed arc 'e' represents the desired exclusion. Via S it receives 1 token upon initialization. All Q functions are assumed to specify a value 1. To be able to study such properties as PT and RF we have included the vertices S and X and arcs s_1, s_2, x_1, and x_2, which serve only to initiate and to terminate the working of the bigraph.
The values of the *L* functions are indicated in the figure.
The set of transformation expressions contains 14 elements:

$a \rightarrow 3;$	$4 \rightarrow x_2;$
$b \rightarrow 4;$	$c,e \rightarrow 1;$
$S \rightarrow s_1,s_2;$	$d,e \rightarrow 2;$
$s_2 \rightarrow 4;$	$3 \rightarrow x_1;$
$3 \rightarrow c;$	$1 \rightarrow a,e;$
$4 \rightarrow d;$	$2 \rightarrow b,e;$
$s_1 \rightarrow 3;$	$x_1,x_2 \rightarrow X;$

With the reduction procedure we can reduce this set of TE's neatly to one single expression: $S \rightarrow X$, which implies that the bigraph is PT.

If both the domain of vertex 3 (indicated as D[3]) overlaps the range of vertex 1 (R[1]), and D[4] overlaps R[2] then the bigraph is also RF.

The CFG for the bigraph in figure 2.20 is relatively complicated. It contains nearly 40 different states. The essence, however, is that one specific state is unreachable from the initial state: state (1,2), which implies that the exclusion is implemented correctly.

Observe that in the bigraph of figure 2.20 arc 'e' is used as an exclusion arc, while all other arcs model ordering relations.

In figure 2.21 we give a bigraph for a deadlock problem. For simplicity we have deleted the inclusion of vertices S and X. The bigraph is not PT (to study PT we must include the S and X vertices). The complete CFG is again quite large. The essence is here that it contains one reachable deadlock state: (1,3).

In figure 2.22 we give a bigraph for a starvation problem. For simplicity we delete vertices S and X and the corresponding arcs again. This time the bigraph is PT. The CFG contains no deadlock states, but it does contain two starvation loops with respect to vertices 2 and 4, or 3 and 1.

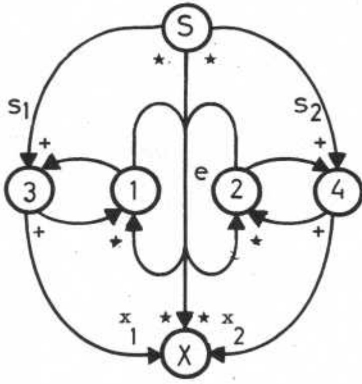


Figure 2.20.
Mutual Exclusion Problem Bigraph

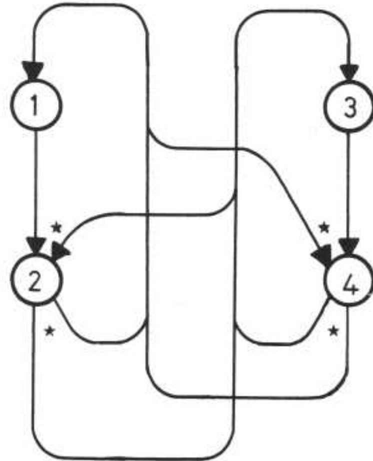


Figure 2.21.
Deadlock Problem Bigraph

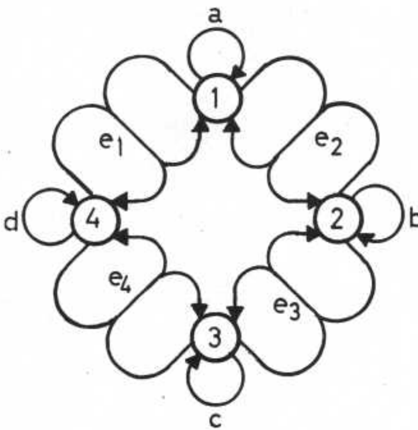


Figure 2.22.
Starvation Problem Bigraph

One of the two loops is given below. (The asterix-superscript denotes a potentially indefinite repetition of the sequence.)

$$\{(1,b,c,d) \rightarrow (1,b,3,d) \rightarrow (a,b,3,d) \rightarrow (1,b,3,d)\}^{\star}$$

It is quite difficult to find a clear bigraph for the solution of the third readers & writers' problem discussed earlier. We again need special negation- and affirmation arcs to make a concise graph. We have derived a complex graph of 18 vertices, which yielded a set of 26 TE's. The set could be reduced to one single TE in five reduction steps, which implies that the graph was PT. A complete CFG is, however, insuperably large, at least for non-automated analysis, and could not be analyzed. For brevity we shall not discuss the graph and the related problems here.

Observations

(1) According to the definitions of a bigraph, the EOR- and AND-logic cannot be mixed on one side of a vertex. This simplifies the bigraphs but also restricts their modelling power. As Cerf remarked this:

> "... may be an unnecessary restriction so far as our analytical tools are concerned. Considering graphs which permit arbitrary initiation and consequence conditions seems a fruitful area, and deserves attention."
> **(Cerf '72, pg. 56).**

(2) The tokens on the arcs of a bigraph can again represent two distinct notions:

- a control-flow point, and
- an execution privilege

(cf. section 2.2.1, Observation 1).
Gostelow wrote:

> "... one may consider resource allocation [cf. **'execution privileges'**] and flow-of-control as different aspects of the same phenomenon, and (-) they, therefore, may be studied as one problem."
> **(Gostelow '72, pg. 753).**

Bigraphs, however, do not allow us to distinguish between these aspects, even if we would like to do so. This has some disadvantages. When we consider tokens as an abstract representation of a condition, for instance, we must take into account that there is a rather strong relation between the evaluation of an initiation-condition, and the effect of that initiation, via the Q^{-} functions.

When a bigraph vertex is initiated *all* (or *one arbitrary*) true precedence condition(s) (depending on the input logic) are/is falsified by removing tokens from the corresponding arcs. Precedence conditions are, however, not necessarily changed by a vertex initiation. In the bigraph model one may thus be forced to make exceptional structures for general cases. This problem is caused by the fact that one makes too many implicit assumptions about the coordination relations that will be modelled. It can, for instance, be extremely difficult to model non-symmetric types of exclusion relations in a bigraph model.

One can attempt to circumvent some of the problems mentioned by introducing dummy vertices or special arcs, like a negation arc or an affirmation arc (which have no 'side-effects' on the conditions evaluated) but the problem remains that conditions cannot be represented directly, nor can they be combined freely. (The same objection was raised for Petri Nets, non-extended Coordination Nets, and Slutz Graphs.)

(3) The property of PT relates to only one specific aspect of tokens in the bigraph: the representation of flows-of-control. The other aspects are neglected. There is no reason, for instance, to require that all 'conditions or 'privileges' should be false on termination of a bigraph. A privilege can only be used via an active flow-of-control, and if the latter has terminated, the privilege is inaccessible anyway.
To study the property of PT we are forced to extend a bigraph with many arcs, which are in fact redundant. The extensions (but not only these) can make a graph very complicated. Cerf noted that bigraphs are sometimes "almost impossible to draw" **(Cerf '72, pg. 106)**.
The provability of PT for uninterpreted (!) graphs is as such still valuable, if only because we do have so few systematic procedures to analyze these graphs.

(4) The use of the complete CFG is, at least for non-automated analysis, doubtful. The number of system states is often insuperably large, which makes analysis by systematic inspection of a complete CFG unreliable.
True enough, the reduction procedures allow one to make a CFG much smaller. However, in these reductions valuable information is lost (e.g. starvation loops disappear).

(5) The property of RF is rather awkwardly defined. We have noted that RF, or repetition freeness, does not imply that a graph is repetition free, but that it *can* be repetition free. It would then be more appropriate to replace the notion of RF by its complement NL, or *n*ecessarily *l*ooping.
The correspondences are: NL = NOT RF and RF = NOT NL.

2.2.5. Coordination Graphs (Belpaire & Wilmotte '73; Belpaire '75)

Description

In Belpaire's model coordination requirements are formalized as binary relations on the set of critical sections. If the pair (i,j) is an element of relation R: $(i,j) \in R$, then the *execution* of section i will prevent the *initiation* of section j. The relation R thus specifies exclusions.
Another relation C can be used to specify orderings or, as Belpaire calls these, 'cooperations'. For each pair (k,l) in relation C we should specify a number $q_{k,l}$ which formalizes that section k should be executed at least $q_{k,l}$ times more often than section l before section l is allowed to initiate.

The coordination requirements specified in relations R and C can be represented in a graph in which each critical section is represented by a vertex. The exclusion relations are represented by directed arcs. Sequencing relations are represented by directed arcs which are annotated with the corresponding "q-numbers". We can best illustrate these ideas with a few examples.

Example 1:

In the second readers & writers' problem (with writer-priority), we have three critical sections: a section for reading, a section for writing, and a section in which writers can wait for access to their writing section. We call these sections, respectively: AR, AW, and WW.
There are 4 pairs in the exclusion relation R:

- (AR,AW)
- (AW,AR)
- (AW,AW)
- (WW,AR)

If "j" is the identification of one specific writer, and WW(j) and AW(j)

represent, respectively, section WW and AW being executed by writer "j", then there are two pairs in relation C, for each writer in the system:

(WW(j),AW(j)), and (AW(j),WW(j)) with $q_{ww(j),aw(j)} = 1$,
and $q_{aw(j),ww(j)} = 0$.

The last q-number specifies that section AW(j) must have been executed at least as often as WW(j) before WW(j) can be reinitiated.
In figure 2.23 we give an exclusion graph and a sequencing graph for this problem.

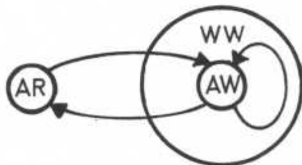


Figure 2.23a.
Exclusion Graph
Second Readers & Writers' Problem

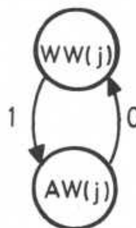


Figure 2.23b.
Sequencing Graph
Second Readers & Writers' Problem

Example 2:

In the producer /consumer problem we have 2 critical sections, named P and C. In relation R we can define 4 pairs:

- (C,C);
- (P,P);
- (P,C);
- (C,P).

If the buffer capacity is N, we can define the following two pairs in relation C:

(P,C) and (C,P) with $q_{p,c} = 1$ and $q_{c,p} = 1 - N$.

The corresponding graphs are given in figure 2.24.

Figure 2.24a.
Exclusion Graph
Producer/Consumer Problem

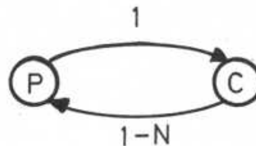


Figure 2.24b.
Sequencing Graph
Producer/Consumer Problem

Belpaire '75 (pg. 41-53) gave systematic procedures with which one can discover deadlocks in Coordination Graphs. We discuss the procedures in the next section.

Application .

A mutual exclusion problem is readily represented in relation R by the two pairs (A,B) and (B,A). Observe that the two other pairs (A,A) and (B,B) are not implicit in this specification (compare this to the Petri Net or Coordination Net specifications). It is readily possible to model nonsymmetric types of exclusion relations. The exclusion graph is given in figure 2.25.
The exclusion graph for a deadlock problem is represented in figure 2.26.
We have drawn two nested pairs of sections: A nested in B, C nested in D.
The exclusion relation R contains two pairs: (B,C) and (D,A). The hierarchical deadlock is clearly visible. Belpaire gave a procedure with which one can establish the existence of a deadlock more formally. Belpaire considered processes with an arbitrary lay-out of critical sections, that is: sections were allowed to overlap each other in any way. If one restricts oneself to processes with only properly nested sections the procedures are much simplified. The latter is more in accordance with the structured programming methodologies (see chapter 3), and we will, therefore, restrict ourselves to just that. To discover deadlocks Belpaire defined, apart from the exclusion and sequencing graphs, a third type of graph which he named the 'critical graph'. In the critical graph one draws a directed edge from section X to section Y if section Y contains at least one subsection Z (on an arbitrary level in the nesting hierarchy) for which the pair (X,Z) is in relation R (that is: X excludes Z).
In figure 2.27 we have given the critical graph for the exclusion graph in figure 2.26. Belpaire showed that if this critical graph contains no cycles the solution is deadlock free. Clearly, the critical graph does contain a cycle: B,D. In this case one must be able to prove that the sections which are included in the cycle (B and D) are 'incompatible', that is: that these sections can never be active simultaneously. It follows from figure 2.26 that the sections B and D are not incompatible at all, and thus the solution does contain a real deadlock.
In figure 2.28 we give the critical graph for the solution to the second readers & writers' problem represented in figure 2.23 a. Clearly, the critical graph contains no cycles, and therefore the solution is deadlock free.

In figure 2.29 we give an exclusion graph for 3 sections A, B, and C.
Relation R contains 4 pairs:

- (A,B)
- (B,A)
- (B,C)
- (C,B)

We can discover a starvation possibility for section B, which can be realized for specific execution speeds of sections A and C. The graph is not very clear at this point. Belpaire gave no procedures to discover starvation loops more systematically.

The formalization of the third version of the readers & writers' problem is not directly possible in Belpaire's model. The model is based on the formalization of static partial ordering relations on the execution of sections, and the third readers'/writers' problem cannot be described (transparently) in such terms (cf. section 2.3.3). Belpaire realized this restriction when the wrote:

> "In some problems, however, the coordination rule can change with the occurrence of particular events and therefore other structures than partial orderings are needed."
> (pg. 12).

> "A general model should, therefore, include the representation of such mechanisms. (-) It, however, falls beyond the possibilities of this thesis."
> (Belpaire '75, pg. 13).

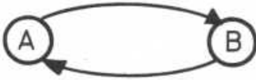


Figure 2.25.
Exclusion Graph
Mutual Exclusion Problem

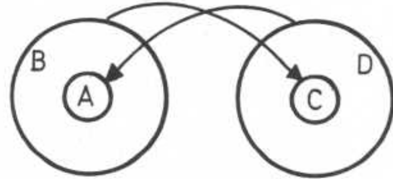


Figure 2.26.
Exclusion Graph
Deadlock Problem

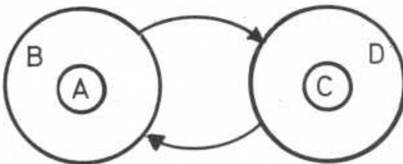


Figure 2.27.
Critical Graph
Deadlock Problem (Fig. 2.26.)

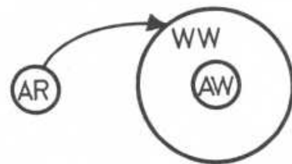


Figure 2.28.
Critical Graph
Second Readers & Writers' Problem (Fig. 2.23.)

Figure 2.29.
Exclusion Graph
Starvation Problem

Observations -

(1) The coordination graphs can be of great help to conceptualize a coordination problem. Problems can be described and analyzed on varying levels of abstraction.

(2) There is a clear correspondence between an exclusion graph and an implemented solution with dependence operations **(Belpaire & Wilmotte '73)**.
A directed edge in an exclusion graph from section A to section B can be 'translated' directly in a d-semaphore DOWN operation at the beginning of the code represented by section A, and a corresponding d-semaphore PASSAGE at the beginning of the code represented by section B.
Thus one can systematically transform implemented solutions to coordination graphs and vice versa.

(3) The restriction to static partial ordering relations, which is implied by the Belpaire model, restricts the modelling power of the Coordination Graphs. We can attempt to relax this restriction somewhat by extending the model such that we can annotate each exclusion relation with a boolean expression which defines the states in which the exclusion relation is applicable. The exclusion relation is then only valid if the corresponding boolean expression is true. In this manner we can describe a solution to the third readers & writers' problem.
The exclusion graph is given in figure 2.30. The arcs from WW to AR, and from WR to AW are annotated with complementary boolean expressions which represent the precedence conditions. The annotations prevent the occurrence of a deadlock. (Compare figure 2.30 to figure 2.26.) It is assumed that state variable 'p' is set to 1 in section AR and reset to 0 in section AW.

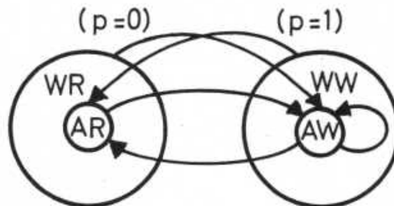


Figure 2.30.
Extended Exclusion Graph
Third Readers & Writers' Problem

(4) We have already noted that one is rather restricted in the types of coordination relations that can be modelled in the (original) Belpaire model. The explicit use of state variables in coordination rules is not readily possible. The access of state variables cannot be visualized. Explicit procedures for the analysis of such items as starvation, reachability and obeyance of precedence rules have not been described (though they are perhaps feasible).

2.3. FORMAL SPECIFICATIONS

In this part we shall discuss a series of more abstract descriptive models in which coordination structures can be formalized in high level specifications. The specifications are independent of specific implementations with standard coordination primitives.

2.3.1. Robinson & Holt Specifications (Robinson & Holt '73; Robinson '75)

Description (part 1)
Robinson and Holt base their method on the software module specification method outlined by **Parnas '72**.

> "We have extended the Parnas specification language to enable specification of solutions for synchronization problems and to use these modules with conventionally specified modules to describe asynchronous cooperating processes."
> **(pg. 2)**.

> "It is intended that [**such a**] specification be amenable to formal verification techniques."
> **(Robinson & Holt '73, pg. 3)**.

The method is to formalize coordination requirements in a set of invariant relations. To 'monitor', and if necessary 'restrict' entries to and exits from sections of code which are considered 'critical' with respect to the observance of the invariant relations, the processes are equiped with abstract *gates*. To enter or to leave a piece of critical code a process must then pass such gates. One abstracts deliberately from the question of how such gates should be implemented, at least in the analysis phase.
A *passage* through a gate is by definition uninterruptable (indivisible). A process can be blocked *in* a gate (prevented from passing through it), but it cannot block still other processes from entering gates while being blocked. The blocked process looses its privilege to execute in a gate the moment it is blocked and suspended. The process regains this privilege when it is awakened and restarted (unless it is restarted not *in* but directly *after* the gate in which it was blocked).

The effect of a passage through a gate on the invariant relations is specified in transition expressions. The essence of the specification method is clearly that no process should be able to pass through a gate if the corresponding effect can invalidate an invariant relation. If a passage would invalidate an invariant the corresponding process must be blocked in the gate until it can proceed without invalidating the invariants.

The invariant relations describe the set of legal system states. There are two types of invariants: *static invariants*, which are independent of the execution history of the system being modelled, and *non-static invariants*.

A complete Robinson & Holt specification includes a formal description of the following items[1]:

- the invariant relations, expressed in
- state variables, parameters, etc.,
- transition expressions, which specify the effects of passing through the gates, and

– – – – –

(1) The terminology of Robinson and Holt is different. **Robinson '75** named the invariant relations as *exception conditions*, state variables as *value functions*, and transition expressions as *operation functions*.

- abstract programs, which consist of program lay-outs with gates.

Examples of the application of this method are given below.

Application

First we consider a mutual exclusion problem. The critical section on which the exclusion is required is equiped with two gates, named 'gate 1' and 'gate 2'. The complete specification of the coordination structure is as follows:

- invariant relation : *max.in.section* ≥ *number.in.section*,
- state variable : *number.in.section* with initial value 0,
- state parameter : *max.in.section*
- transition expressions: *gate 1*: *number.in.section +:= 1*[1]
  gate 2: *number.in.section −:= 1*
- abstract program : *gate 1; critical section; gate 2.*

In the above example we defined one invariant relation: a static invariant. In the following example we have 3 static invariants and one non-static invariant.

The problem we consider is the second version of the readers & writers' problem (writer priority). In the reader program we include two gates: 'gate 1', and 'gate 2'. In the writer program we include 3 gates: 'gate 3', 'gate 4', and 'gate 5'. The complete Robinson & Holt Specification is given below.

- invariant relations: (1) $0 \leq AR \leq R$;
  (2) $0 \leq AW \leq 1 - min(AR, 1)$;
  (3) $0 \leq WW \leq W$;
  (4) $min(AR - AR', 1) \leq 1 - min(WW, 1)$.

The fourth invariant relation is a non-static one. The symbol AR' represents the value of variable AR before it is changed by the effect of the passing of the gate in which it is evaluated. The non-superscripted variables denote the value of the corresponding state variables which they would have *after* such a change.

- state variables : AR, AW, and WW all with initial value 0,
- state parameters : R and W (indicating the number of readers and writers, respectively),
- transition expressions: *gate 1*: AR *+:= 1*
  gate 2: AR *−:= 1*
  gate 3: WW *+:= 1*
  gate 4: WW *−:= 1*; AW *+:= 1*
  gate 5: AW *−:= 1*.
- abstract programs : *gate 1; read; gate 2.*
  gate 3; gate 4; write; gate 5.

More complicated still is the formal specification of the solution to the third version of the readers & writers' problem. There are 4 static invariants, and 2 non-static invariants, 5 state variables, 2 parameters, and 6 gates:

- invariant relations : (1) $0 \leq AR \leq R(1 - min(AW, 1))$
  (2) $0 \leq AW \leq 1 - min(AR, 1)$
  (3) $0 \leq WW \leq W$
  (4) $0 \leq WR \leq R$
  (5) $min(AR - AR', P, 1) \leq 1 - min(WW, 1)$
  (6) $min(AW - AW', 1 - P, 1) \leq 1 - min(WR, 1)$

— — — — —

(1) We abbreviate the expression "x := x + 1" here as "x *+:=* 1", and similarly "x := x - 1" as "x *−:=* 1".

- state variables : *AR*, *AW*, *WR*, *WW*, *P* all with initial value 0. Variable *P* represents the preference: *P* = 0 means 'reader preference', *P* = 1 means 'writer preference'.
- state parameters : *R*, *W*.
- transition expressions: *gate 1*: *WR* +:= *1*
  gate 2: *AR* +:= *1*; *WR* −:= *1*
  gate 3: *AR* −:= *1*; *P* := *1*
  gate 4: *WW* +:= *1*
  gate 5: *AW* +:= *1*; *WW* −:= *1*
  gate 6: *AW* −:= *1*; *P* := *0*
- abstract programs : *gate 1*; *gate 2*; *read*; *gate 3*.
  gate 4; *gate 5*; *write*; *gate 6*.

Description (part 2)
Robinson '75 extended the model by assigning two implicit variables to each gate: *through(gate)* and *waiting(gate)*.
The variable *through(gate)* indicates the number of processes that have passed the gate indicated. The initial value of *through(gate)* is zero.
The variable *waiting(gate)* indicates the number of processes that are blocked in the gate indicated. The initial value is zero.
With these *gate variables* we can simplify the formal specifications.
For instance, in the example of the third readers & writers' problem we can omit gates 1 and 4, and all explicit state variables except *P*.
AR becomes *through(gate 2)* - *through(gate 3)*,
WR becomes *through(gate 5)*, etc.
Observe that a process cannot be blocked when leaving a critical section:
waiting(gate 3) = *waiting(gate 6)* = *0*.

To analyze a system of formal specifications one can partition the set of system states in two subsets: desirable (or permissible) states and undesirable (or prohibited) states. It must then be verified that the invariant relations formalize precisely the subset of desirable states. It must be verified that each desirable state is reachable, with the given transition expressions, invariants, and abstract programs.

To verify whether the system is deadlock free it must be checked whether there is a sequence of state transitions which can lead from each reachable state to any other permissible state. To analyze these properties one can use a reduced system transition diagram (comparable to a bigraph CFG). We will exemplify these notions with the aid of the formal specifications of the second readers & writers' problem presented above.

Robinson & Holt '73 defined the following 6 permissible (macro) system states, in terms of the state variables AW, AR, and WW.

	AR	AW	WW
(a)	0	0	0
(b)	⩾1	0	0
(c)	⩾1	0	⩾1
(d)	0	1	0
(e)	0	1	⩾1
(f)	0	0	⩾1

Note that states b, c, e, and f represent subsets of the set of system states which yields the reduction in the size of the transition diagram. The corresponding (reduced) system diagram is given in figure 2.31. The annotations on

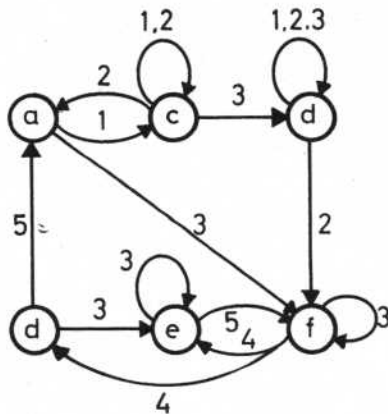


Figure 2.31.
Reduced System Diagram
Second Readers & Writers Problem

Legend:
1 = start-reading,
2 = finish-reading,
3 = request-writing,
4 = start-writing,
5 = finish-writing.

a = empty state,
b = readers active,
c = readers active and write-request(s),
d = writer active,
e = writer active and read-request(s),
f = write request(s).

the transition arcs in figure 2.31 refer to gate numbers. The annotations in the vertices refer to the states as defined in the table.
With the aid of the system diagram one can verify that the solution is deadlock free. Observe that there is a possibility that the reduction introduces spurious paths[1] in the system diagram, which may complicate the analysis. (Robinson & Holt have not discussed this problem.)

Observations

(1) Robinson & Holt have based their model explicitly on systematic abstraction methods, like the Parnas module specification procedures. Their intent has been to design a model in which one can distinguish computations from coordination, in the design and analysis of coordination problems. In this respect their specification technique is an important contribution to the theory of process coordination.

(2) A first point of criticism to the Robinson & Holt Specifications is that the non-static invariants in their model do not really formalize properties of *states* but rather properties of *behavior* **(Greif '75)**. As such this type of invariant relation is defined on the wrong level of abstraction. Properties of behavior can best be defined on the level of the processes, and not on the level of the system of processes.
Greif has also questioned the use of the *min* and *max* functions in high level specifications. **(Greif '75, pg. 94, footnote)**.
The invariant relations are sometimes quite difficult to discover, and Greif is certainly right when she states that invariants such as invariant (4) in the second readers & writers' problem, or invariants (5) and (6) in the third readers & writers' problem, lack descriptive clarity.

(3) A second point of (mild) criticism is that Robinson & Holt did not present or discuss conversion rules which will transform a specification into an implementation with standard coordination primitives.

(4) Finally, the analysis methods described by Robinson & Holt are rather restrictive. Items like starvation and the obeyance of complex preference rules, for instance, are not analyzed. It is not discussed how one can reduce system diagrams without introducing spurious paths.

2.3.2. The Actor Model (Greif '75; Goodman '76; Greif '77)

The 'actor model of computation' introduced by **Hewitt '73, '74**, was applied to the study of coordination problems by Greif and Goodman. Greif developed a formal specification language for coordination problems, much in line with the Robinson & Holt Specifications considered earlier. A brief description of the actor model will precede the description of the specification language.

Description of the Actor Model

All 'computational entities', like data, operations and procedures, are called *actors*. Information is passed between actors via *events* called *message transmissions*. An example of such events is the passing of the argument to a procedure, or the returning of the outcome of a procedure. A message transmission is a directed event from a *source* to a *target* (both actors). An actor is defined by the responses which it generates to the messages which it receives. A specific actor may respond to only a specific type of message. For instance, the actor 'addition' receives a message with 2 arguments (say N and M) and returns a message with one argument (N + M). If the contents of a message are not acceptable to an actor, the actor will return a special error-message.

— — — — —

(1) See **Howard & Alexander '73, pg. 241**.

Message transmission are formalized as follows. We give the notation suggested by Goodman; Greif's notation[1] is less transparent.

<*target* receives (message: *argument* (reply to: *cont.*)) {in α, *ec*}>

in which:

- *target* : is the name of the target actor (e.g. 'addition');
- *argument*: is a sequence of arguments sent to this target actor (e.g. N, M);
- *cont.* : is the name of the actor to which a return message should be sent (the return message of the addition actor would be N + M; *cont.* is short for 'continuation');
- α : is the name of the *stream* of events in which the current message transmission occurs. Each event can cause another event via the continuation. A stream of events can be interpreted as a process. In the actor model the process is indicated as an *activator*. The specification of the stream is optional.
- *ec* : is the event count of the activator specified in α (cf. an instruction counter in a sequential process). The specification of the event count is again optional.

The words 'receives', 'message', 'reply-to', and 'in' are key-words.
The invocation of the addition-actor would be formalized as follows:

<*addition* receives (message: *2*, *3* (reply-to: *cont.*))>
<*cont.* receives (message: *5* (reply-to: *?*))>
...

A specific ordering of events is called a *behavior*.
A sequential process can be interpreted as a completly ordered set of events.
Concurrent processes then correspond to partially ordered sets of events.

Actors can be defined on varying levels of abstraction. A description on a high level of abstraction can be used either as a formalization of actual behavior in a system, or as a formal specification of intended behavior in the design of a system.

The actor model can be further extended with so-called 'cells of read/write storage' with axiomatic properties, and with primitive operations for the creation of activators.
It is required that a single actor may only generate one single message in a stream. This restriction serves more to simplify the model than the descriptions in this model, and therefore we will not elaborate it here.
In the actor model one can express sequential execution, jumps, coroutines, and concurrency; however, selection and iteration structures are not trivially covered. A detailed discussion of the actor model falls beyond the scope of this study. The outline given above will suffice to follow the discussion below.

Description of the Specification Language

Greif uses an abbreviated notation for events, with the subscripted symbols E and S. The E is used to represent events, the S is used to represent sequences of events. For instance:

$$S_1 = (E_1, E_2, \ldots, E_k).$$

Orderings of events are indicated with two symbols: $\dashrightarrow$ and $\Rightarrow$.
The first ordering symbol is used for direct continuation links between events in the same activator. The second symbol is used to indicate a *time-ordering*.
A time-ordering can be realized by a sequence of direct continuation links,

(1) Greif used the notation: <*target* (apply: *argument*(then-to: *cont.*)) α, *ec*>

and/or the application of so-called 'causality axioms' (we shall return to this later).
The ordering property "event 2 is only executed if event 1 has completed" can then be formalized with a time ordering, as:

$$E_1 \Rightarrow E_2.$$

Mutual exclusion between these two events can be specified as:

$$E_1 \Rightarrow E_2 \quad \text{OR} \quad E_2 \Rightarrow E_1.$$

As Greif stated, this is essentially a *behavioral* approach to the specification of coordination problem solutions, which can be contrasted with the state invariant approach of Robinson & Holt.

Greif has given some examples of formal problem specifications in this language. Formally, the specifications are sentences in first-order predicate calculus **(Greif '77)**. The application of the language in its basic form presents severe modelling problems. Greif solves these by including special actors (operators) in the model, with axiomatic properties. The actor 'protected data base' is introduced, for instance, with the axiomatic property that it orders all read and write operations which are directed to it, in time. The read and write operations may be higher level operations, consisting of sequences of message transmission (e.g. destructive-read; rewrite). These higher level operations are then ordered such that the sequences of events from which they consist will never overlap.
Within the protected data base one can distinguish between *event requests* and *event executions*. (Observe that this is not possible in the original actor model.) Requests are represented with a subscripted symbol R; executions with a subscripted symbol E, or S (the latter for sequences). (Greif did not use the symbol R.)

Application
Above we have indicated how one can specify a mutual exclusion relation with a simple time ordering. To describe a non-symmetric exclusion relation is, however, much more complicated, as we shall see shortly.

First we consider a specification of the solution of the second readers & writers' problem (with writer priority).
We will use the following symbols:

- R_r : denotes a reader access-request event (to the protected data base);
- $E_{r\text{-}in}$: denotes the first event in a read operation;
- R_w : denotes a writer access-request event;
- $E_{w\text{-}in}$: denotes the first event in a write operation;
- $E_{r\text{-}out}$: denotes the last event in a read operation;
- $E_{w\text{-}out}$: denotes the last event in a write operation;
- S_r : the sequence $(E_{r\text{-}in}, \ldots, E_{r\text{-}out})$, always preceded by R_r;
- S_w : the sequence $(E_{w\text{-}in}, \ldots, E_{w\text{-}out})$, always preceded by R_w.

E and *S* denote arbitrary events, respectively, sequences.

Formal specification:

(1) Readers exclude writers:

$$(\forall S_r)\ (\forall S_w)\ (S_r \Rightarrow S_w \text{ OR } S_w \Rightarrow S_r);$$

(2) Writers exclude writers:

$$(\forall S_{w_1})\ (\forall S_{w_2})\ (S_{w_1} \Rightarrow S_{w_2} \text{ OR } S_{w_2} \Rightarrow S_{w_1});$$

(3) No starvation, no deadlock:

$$(\forall R_r)\ (\forall R_w)\ (R_r \supset S_r \text{ AND } R_w \supset S_w).$$

The symbol ⊃ denotes that the occurrence of the event specified on the left-hand side *implies* the occurrence of the event specified on the right-hand side, within finite time.

(4) Precedence rule for write requests:

$$(\forall R_{w_1})\ (\forall R_{w_2})\ (R_{w_1} \Rightarrow R_{w_2} \supset S_{w_1} \Rightarrow S_{w_2}).$$

(5) Writer priority: $(\forall S_r)$

$$((S_r) \supset ((\exists E)\ (E \Rightarrow R_r \text{ AND}$$
$$(\forall R_{w_1})\ \neg(R_{w_1} \Rightarrow E \text{ AND } E = E_{w\text{-out}} \text{ AND}$$
$$(\exists R_{w_2})\ (R_{w_2} \Rightarrow E \text{ AND } R_{w_2} \neq R_{w_1} \supset E_{w_2\text{-out}} \Rightarrow E)) \text{ AND}$$
$$\text{NOT } (\exists R_{w_3})\ (E \Rightarrow R_{w_3} \Rightarrow R_r)))).$$

Informally, the fifth property states that the occurrence of S_r implies that at the time of the execution of E_{r-in}, no writer is executing and no writer has requested access to the data base and is waiting to initiate.
Extending this specification to the solution of the third readers & writers' problem requires the formalization of such notions as 'the last executed event in the protected data base', 'the number of waiting readers', etc. This leads to a drastic increase in the size and complexity of the formal specifications. We will not attempt to elaborate the extension here.
Greif '75 investigated a similar version of the problem (discussed earlier by **Hoare '74**), and is forced to use statements like:

"... the first unserved write request preceding E ...", and

"... either it is a writers' 'turn' at E and there are no writes before E_w which are waiting, or E 'ends a write' and there are no reades before it or writes before E_w which are waiting."
(Greif '75, pg. 71).

The transparency of such a specification is, in our view, minimal.

Observations

(1) We have already outlined some notational difficulties which undermine the applicability of the actor model to the specification of coordination structures. More serious still is that the specification language does not allow us to formalize general delay conditions, as they are used in conditional critical regions, monitors, or extended dependence operations.
One problem, with respect to the modelling of monitor-like structures, is in the formalization of signal releasing operations and of the causal link between signal and wait operations. Greif noted this flaw when she wrote:

"... specifying exactly which event E [**corresponding to a signal**] is required is somewhat complex."
(Greif '75, pg. 85).

Quite remarkably, however, Greif then concludes that:

"... this part of the 'structured monitor' can only be considered a very unstructured synchronization primitive."
(Greif '75, pg. 85).

In our view the incapacity of the actor model, and the specification language based on it, to represent general synchronization conditions concise-

ly is a severe restriction to its applicability.

(2) The 'behavioral approach', introduced by Greif, does not lead to simpler specifications, or specifications which are easier to analyze and verify, than for instance the state-invariant approach introduced by Robinson & Holt. The following statement in Greif's thesis shows that Greif misinterpreted the state invariant approach on these points:

> "If the [**coordination**] properties are based on sequences of state changes longer than one change, the Robinson & Holt specifications may not be at all useful."
> **(Greif '75, pg. 94).**

It is not difficult to show that an invariant expression in the language of Robinson & Holt, is always based on at most 1 state change, namely 'the next'. All previous state changes, in so far are they are relevant to the enforcement of the coordination rules, can be *abstracted* with the use of state variables and state parameters. The state invariant approach derives its power just from these possibilities for the abstraction of entire execution histories in the of only a finite set of state variables. The behavioral approach lacks this abstraction power, and we therefore have strong reasons to question Greif's statement that:

> "... to write specifications dependent on behaviors that correspond to long sequences of 'state transitions' (-) the behavioral approach is simplest and most direct for implementation independent specifications."
> **(Greif '75, pg. 95).**

With the behavioral approach one specifies an intended behavior in event execution histories. With a state invariant approach one can abstract from such histories and concentrate on states. That the latter approach can lead to more transparent descriptions can easily be verified by comparing the formal specification of the second readers & writers' problem of Robinson & Holt (see page **102**) with the one given by Greif (page **107-108**). Compare, for instance, invariant relation (4) (page **102**) with the corresponding statement in Greif's language, specification (5) (page **108**).

(3) The emphasis that is laid on partial orderings in Greif's model is questionable. As Belpaire correctly states:

> "In some problems (-) the coordination rule can change with the occurrences of particular events and, therefore, other structures than partial orderings are needed."
> **(Belpaire '75, pg. 12).**

(4) We have noted that the conceptual clarity of the specifications in Greif's model is often insufficient. We subscribe to Greif's frank remark that:

> "... the formal language definition serves only the function of formality and will add no insight into parallel processing."
> **(Greif '75, pg. 18).**

(5) Final points of criticism can be that there are no systematic conversion methods from specifications to solutions with standard programming primitives, no systematic verification procedures described with the model, etc.

(6) **Atkinson & Hewitt '76** have extended the work of Greif by defining still another primitive actor in the model, called the *serializer*.
In their words:

"... a serializer is defined by the relationships among the events which are caused by [it] ."
(Atkinson & Hewitt '76, pg. 267).

The serializers are used to enforce the obeyance of coordination rules.

"A serializer should be constructed to surround the protected resource in such a way that it is impossible to accidently avoid passing through it when using the protected resource."
(ibid., pg. 268).

As an axiomatic property of serializers it is stated that processes exclude one another from the possession of the serializer body. (Compare with the exclusion on monitor-procedures.)
The serializer differs from the standard monitors, among other things, in that it has no signal-type operations. (Most likely because they present modelling problems, as was noted above (point 1)).
Local to the serializer body are *queues* and *crowds*. The queues are used to delay processes in; the crowds are used to identify the processes which are executing on critical resources (mainly for debugging purposes).
In the serializer one can explicate delay conditions by means of *guarantee statements*.
One example of a serializer solution to a simple mutual exclusion problem is given below:

```
(cons.serializer
      (queues: Q) (crowds: C)
      (entry:
            (enqueue Q (guarantee: (empty: C))
                  (then: (relay-to: resource (thru: C))))))
```

(See remark at point 4.)

2.3.3. Path Expressions (Campbell & Habermann '74; Habermann '75; Flon & Habermann '76; Lauer & Campbell '75)

Campbell & Habermann introduced a method to specify coordination requirements formally in so-called *path expressions*. The path expression specifications can be incorporated in the declaration of abstract data structures. This prompts a direct implementation of path expressions in programming primitives.
Here, we consider the path expressions as formulas in a specification language, and discuss their modelling power.

A class of abstract data 'objects' can be formalized in a *data type*.

"A type definition describes (-) the possible states of an object of that type and the ways in which it can be manipulated."
(Flon & Habermann '76, pg. 141).

For instance, one can define a type *set* with two legal (abstract) operations (manipulations) called *set-addition* and *set-substraction*.
The way in which the abstract operations are implemented is hid from the user. The user knows only the effect of the operation, not how the effect is realized. One uses fundamental notions from structured programming here, namely: data abstraction and information hiding. Flon & Habermann discussed the declaration of both operations and procedures. In the following discussion we restrict ourselves to operations only.

The set of legal abstract operations is specified in data declarations.

The path expressions are used to restrict the order in which these abstract operations can be applied in user processes. More precisely:

> "The path expression defines the set of legal sequences in which the operations of a type can be applied to an individual object of that type."
> **(Flon & Habermann '76, pg. 141).**

A simple example of a path expression for the three abstract operations *send*, *receive*, and *remove* is:

path *send*; *receive*; *remove* **end**.

For simplicity the name of an abstract operation may occur only once in a path expression. Path expressions may not be nested.

The symbol ";" is a *path expression operator*, which is called the *sequencing operator*. The operands of a sequencing operator are to be executed one by one, from left to right. The key-word brackets "path" and "end" indicate a cyclic repetition. (The operations listed may be applied in the specified order repeatedly.)
Other path expression operators are:

- "," or "+": for exclusive selection;
- "*" : for indefinite repetition (to be used as a superscript on a bracketed part of the path expression).

A pair of braces "{...}" around a (series of) abstract operation(s) indicates that the given operation(s) may be initiated by any number of concurrent processes at a time. The initiation of the part enclosed in the braces is defined as the first initiation of the first operation specified; the termination is defined as the first termination of the last operation specified, whereafter no process is executing any of the operations specified.

A special path entry is the *numeric element*. As an example, the numeric element of the three operations *send*, *receive* and *remove*

$$(send - receive - remove)^n$$

would indicate that:

$$|\text{send}| \geq |\text{receive}| \geq |\text{remove}| \geq |\text{send}| - n$$

where $|\text{send}|$ indicates the number of times that operation 'send' has been executed **(Flon & Habermann '76, pg. 147).**
A path expression specifies exclusions implicitly. Unless the operations are enclosed in braces, as described above, their execution is assumed to be indivisible.
If the execution of an abstract operation is requested by a user process, at a time when this execution would violate a path expression, the requesting process is delayed in a sleep-wait until it can proceed without violating any path expression. The termination of an abstract operation signals all sleeping processes which are able to proceed implicitly.

Application
A fairly straightforward application of path expressions is possible for the first version of the readers & writers' problem (reader priority):

path {read}, write **end**.

(Remember the meaning of the colon and of the braces.)
For the second readers & writers' problem (writer priority) we arrive at 2 path expressions:

(1) **path** request-read, {request-write; write} **end**;
(2) **path** {request-read; read}, write **end**.

(Campbell & Habermann gave a slightly different solution.)
The read and write abstract operations, as they are available to users, should then be constructed as follows:

READ = **begin** request-read; read **end**
WRITE = **begin** request-write; write **end**.

The first path expression specifies that request-read operations are mutually exclusive, and that once a request-write operation has been initiated, no new request-read operation may be started. After zero or more read operations (of all readers which have previously executed their request-read operation) the second path expression will allow precisely one writer to initiate and execute. One by one all writer requests can then be served, without the possibility of a reader request interfering.
The extension of this specification to a solution for the third readers & writers' problem seems, at first sight, surprisingly simple. We merely change the second part of the first path expression into a 'closed block', with brackets:

(1') **path** request-read, (request-write; write) **end**;

Observe, however, that the fairness of the solution depends on the implementation of the exclusive selection operator ",".
When we 'translate' the path expressions to a solution with dependence operations, we obtain the following programs:

```
WR: WW: down WR;                 WW: WR: down WW;
        request-read;                    request-write;
        up WR down AR;               AR:
        read;                            write;
        up AR.                           up WW.
```

The distinction between d-semaphore WW and AW has become irrelevant, therefore, the latter semaphore has been eliminated from the above solution.
Whether the requirement is obeyed that no process may be starved (neither readers nor writers) depends on the implementation of the signalling and waiting strategies.
It is much more difficult to find a solution in which the obeyance of the starvation property is explicated (not implementation dependent).
In an attempt to solve this problem we studied the application of path expressions to a simpler mutual-exclusion-without-starvation problem, for two concurrent processes. These processes can be divided in three subsections: request, access, and leave. If we name the two processes A and B, these subsections can be represented by the symbols: RA, EA, XA, and RB, EB, XB, respectively.
A complete path expression can be found with the aid of a system diagram.
We found the following description:

```
path ((RA,(RB;RA;EB;XB),(RB;EB;RA;XB));((EA;XA),
       (((RB;EA;XA),(EA;RB;XA));((RA;EB;XB),(EB;RA;XB)))*,
       (((RB;EA;XA),(EA;RB;XA));EB;XB)),
      (RB,(RA;RB;EA;XA),(RA;EA;RB;XA));((EB;XB),
       (((RA;EB;XB),(EB;RA;XB));((RB;EA;XA),(EA;RB;XA)))*,
       (((RA;EB;XB),(EB;RA;XB));EA;XA))) end.
```

In spite of the fact that we have restricted this problem to only 2 competing processes, the path expression representation is quite complex. The extension to the complete third readers & writers' problem would make the description still more complex, and is therefore not elaborated here.

Observations

(1) The solution to the third readers & writers' problem based on the straightforward representation in path expressions, given above, could be constructed with fewer operations and simpler coordination tools than the solution derived earlier with extended dependence operations (chapter 1, section 1.3.2.3.2). The new solution is, however, more restrictive, due to the additional exclusions on the request-read and request-write sections.
The new solution is also less transparent than the original one, due to the fact that the obeyance of some coordination properties, like the obeyance of the 'no-starvation' property, have been made an implicit part of the implementation of the coordination tools. This solution gives no clear representation of the problem.
Most likely there is a simpler Petri Net, Slutz Graph, Bigraph, Robinson & Holt Specification, and Actor Model representation of the new solution, compared to those discussed for the solution derived earlier. It is quite remarkable that all these modelling tools yield highly complex descriptions for conceptually clear programs (e.g. with explicated delay conditions) and *qua structure* simple descriptions for programs which are less transparent. These models, then, seem to require (in modelling for analysis) or to yield (in modelling for design) other program structures than those which are conceptually most simple.

(2) **Flon & Habermann '76** stated that:

> "Path expressions provide a powerful additional verification tool."
> **(pg. 141)**.

In fact, it can be quite difficult to analyze the working of a given path expression, as can be seen for the examples given.

(3) The path expressions have the advantage of their structuredness. As Flon & Habermann have noted:

> "... synchronizing conditions are localized in a concise syntax and are not part of the individual operations as they are with monitors."
> **(pg. 142)**.

Still, we object to the view that:

> "Synchronization is a global property and should therefore be specified globally."
> **(pg. 142)**.

Process coordination requires the formalization of a complex of inter-process relations. The inter-process relations are best described on the level of the processes, and not on a system's level (as a global property). Furthermore, the path expressions are based on the formalization of behavioral properties, and they can only represent states via a detour (cf. the numeric elements). We have considered the disadvantages of a behavioral approach in the preceding section (2.3.2).

(4) **Lauer & Campbell '75** described a method to transform path expressions into Petri Nets. These Petri Nets can be used in a correctness analysis.
The transformations yield fairly complicated nets. An example elaborated by Lauer & Campbell of a solution to the cigarette smokers' problem yielded a net of some 45 places, 14 transitions, and 150 transition arcs. Such large nets can add little clarity, at least for manual correctness analysis.

(5) One may conjecture that the obeyance of such coordination properties as 'absence of starvation and deadlocks' belong to the implementation level and should never be explicated in a solution (cf. **Lamport '76**).
In our view this does not solve the modelling problems we have encountered here (see point 1), but only moves them to another level. Furthermore, the choice of placing specific concerns on different levels should not depend on the modelling power of our analysis tools, as it does in the present case, but must be 'open' to the programmer.

2.4. MODELS BASED ON FIRST ORDER PREDICATE CALCULUS

In this chapter we restrict our discussion explicitly to the models that are available for the analysis of coordination problems in multiprocessing systems. The models we are about to discuss are, however, heavily based on specific correctness analysis methods for the computations performed by *sequential* processes. Some background information on the origination and evolution of these methods is therefore in order.

2.4.1. Outline of analysis methods for sequential programs

The desirability of more formal verification techniques for sequential programs was mentioned already by John von Neumann (see **Taub '61**). Later, John McCarthy discussed the item at the IFIP congresses of '62 and '65. The matter gained only a wider interest, however, after the publication of the articles of Naur in 1966 and Floyd in 1967. Naur and Floyd were the first to describe explicit techniques for proving the correctness of programs.

We will not attempt to give an extensive treatment of the existing correctness proving techniques. Such treatments can be found elsewhere. (See e.g. **Elspas** *et al.* **'72** and **Manna & Waldinger '78.**) Here we restrict ourselves to indicating some major lines in the development of the techniques. As a guide-line we use a classification in: an informal approach **(Naur '66)**, the Floyd/Hoare logic **(Floyd '67; Hoare '69)**, a constructive approach **(Dijkstra '68)**, and methods based on uninterpreted program schemas.

Informal approach **(Naur '66)**
In his article of '66 Naur introduced an intuitive approach to program verification. Naur based his method on a technique in program testing in which relevant parts of a data base are 'dumped' at specific points during program execution. By evaluating such 'intermediary results' one can attempt to discover errors in a program. Clearly, the intermediary results are not completely reliable, as (in Naur's words) they:

> "depend entirely on the choice of the data set."
> **(Naur '66, pg. 313).**

Naur named the intermediate dumps of data values 'snapshots'. To verify the correctness of a program, Naur suggested using more 'general snapshots', which would be valid independent of specific data values.
These 'general snapshots', or 'assertions' should express relations between data values, which must hold every time the corresponding points in the program text are reached during an execution. These 'general snapshots' are therefore sometimes described as 'invariant assertions' **(Manna & Waldinger '78).**
The choices of the assertions and of the points to which they refer are entirely up to the programmer. One can assign one such assertion which formalizes the initial state, to a point directly before the first executed statement, and another, formalizing the desired result of the computation, to a point directly after the last executed statement. One can then prove that the last assertion is implied by the first (and the code in between) in a finite number of steps, by attaching similar assertions to intermediate points in the program code.

The advantages of that approach were summarized by **Elspas '72 (pg. 115)**, as follows:

> "1. the verification process is broken up into proofs of several smaller theorems, each corresponding to a program path;
> 2. if a particular implication between a given pair of assertions

is shown not to hold, the programmer is alerted to a particular section of the program for a possible error; and

3. in the synthesis of a program, the development of the (-) assertions prior to the writing of the code will tend to produce modular easily understood programs."
(Elspas *et al.*, **'72, pg. 115).**

The implications between the intermediate asserions are proven by disciplined reasoning and mathematics. The invariant truth of the assertions which occur in loops can be established by mathematical induction.
Manna & Waldinger summarized the verification process as follows:

> "We have thus transformed the problem of proving the partial correctness of programs to the problem of proving the truth of several mathematical theorems.
> The verification of a program with respect to the given input-output assertions consist of three phases: finding appropriate intermediate assertions, generating the corresponding verification conditions, and proving that the verification conditions are true."
> **(Manna & Waldinger '78, pg. 203).**

The 'verification conditions' formalize under what conditions an implication between intermediate assertions will hold. The method described can be used to prove that a program will satisfy the output assertions for any legal input, *provided* that the program will terminate. Manna & Waldinger, therefore, used the term 'partial' correctness. A proof of 'total' correctness requires also a proof that the program will indeed halt for all legal inputs.
A straightforward method to prove program termination is to associate with each loop a new variable called a 'counter'.

> "The counter is initialized to 0 before entering the loop and incremented by 1 within the loop body. We must also supply a new intermediate assertion at a point inside the loop, expressing that the corresponding counter does not exceed some fixed bound. In proving that the new assertion is invariant, we show that the number of times the loop can be executed is bounded.
> Once we have proved that each loop of the program can only be executed a finite number of times, the program's termination is established."
> **(Manna & Waldinger '78, pg. 211).**

A disadvantage of the intuitive proofs is that they tend to be long and untransparent even for trivial programs. The primary goal of program verification, which is to raise the confidence level of a program, is thus often not fully realized.
As Mills worded it:

> "The persuasion of a proof depends not only on its formality, but on its brevity."
> **(Mills '72, pg. 6).**

It may further be quite difficult to find the right assertions to verify a program:

> "...to find them one must understand the program thoroughly."
> **(Manna & Waldinger '78, pg. 204).**

Elspas '72 even criticized all non-automated verification methods as they are:

> "...as fallible as the person carrying out the proof."
> **(Elspas '72, pg. 120).**

Floyd/Hoare logic, or the Inductive Assertion Method **(Floyd '67; Hoare '69)**
According to a statement of Naur's **(Naur '66, pg. 316)** Floyd's proof methods were developed independently, and are not based on Naur's intuitive methods. Still, the Floyd/Hoare logic can well be interpreted as an extension of Naur's technique.
Floyd calls the 'general snapshots' or 'assertions', *propositions* or *tags*. The important difference is that Floyd requires that propositions should be associated with every statement in a program. Floyd then formulates general axioms for the manipulation of propositions, derived from first order predicate calculus. With special semantical definitions Floyd defined the effect of assignments, two control flow structures (branch and join) and the 'start'/'halt' commands. With such axioms one can prove *partial* correctness. To establish *total* correctness Floyd suggested giving a proof of termination with a method similar to the one described above.

> "... by showing that each step of a program decreases some entity which cannot decrease indefinitely."
> **(Floyd '67, pg. 19)**.

The model introduced by Floyd and elaborated by Hoare can be formalized in axioms and rules of inference. The notation most commonly used today was introduced by **Hoare '69.** In this notation, the statement:

$$P \{Q\} R,$$

in which P and R are propositions (assertions), is interpreted as follows: 'If assertion P is true before the initiation of statement (or program segment) Q, then assertion R will be true on its completion, if the execution of Q terminates at all.'

Gries '77 gives two axioms which formalize the effect of, respectively, a null-statement and an assignment:

$$A(0) \text{ for any } R: \quad R \ \{skip\}\ R \ \text{ holds},$$
$$A(1) \text{ for any } R,\ x \text{ and } e: \quad R^x_e \ \{x{:=}e\}\ R \ \text{ holds},$$

where R^x_e represents the result of the substitution of expression e for every free occurrence of x in R.
Axiom A(1) specifies an *antecedent* assertion R^x_e of an assignment operation, when given the *consequent* assertion R.

Floyd originally suggested working in the other direction, which is less clear though. Floyd used the complement of A(1) which is:

$$A(1)' \text{ for any } P,\ x \text{ and } e: \quad P \ \{x{:=}e\} \quad (\exists\, x_0\ (x = S^x_{x_0}(e) \text{ AND } S^x_{x_0}(P))) \text{ holds}.$$

$S^x_{x_0}$ represents the result of the substitution of x_0 for every occurrence of x in expression e, after systematically changing bound variables of e to avoid confusion with free variable x_0 **(Floyd '67)**.
Floyd thus derived a consequent assertion $S^x_{x_0}(P)$ from the antecedent P.

One can construct two symmetric (or 'complementary') verification models in this manner: one with backward inferences, and one with forward inferences **(Sintzoff '74)**.
Working with forward inferences one tries to derive the *strongest* consequent from a given antecedent, and in the opposite case, working backwards one can try to derive a *sufficient* pre-condition from a given consequent **(Hoare '69)**, or even better: a *necessary and sufficient* or *weakest* pre-condition from a given consequent **(Dijkstra '73, '76)**.

Pratt '76 compared this duality in the Floyd/Hoare logic with the duality between 'true' and 'false' in conventional ('static') logic. The logical *modus tolens*, for instance, transforms in the Floyd/Hoare logic from:

$$(a \supset b) \equiv (\text{NOT } b \supset \text{NOT } a)$$

into:

$$P \{Q\} R \equiv \text{NOT } R \{Q^-\} \text{ NOT } P$$

where Q denotes a forward execution, and Q^- denotes a backward 'execution' of a statement or program segment. (Instead of 'backward execution of Q' one can speak of a 'backward inference over Q'.)

The inference rules for the three standard control flow structures can be notated, independent of backward or forward inferences, as follows (ALGOL 68 notation):

I(0) **concatenation**, for any P, Q_1, Q_2, S and R:

$$P \{Q_1\} S \text{ AND } S \{Q_2\} R \Rightarrow P \{Q_1;Q_2\} R$$

I(1) **selection**, for any P, B, Q_1, Q_2, S, and R:

$$(P \text{ AND } B \{Q_1\} R) \text{ AND } (P \text{ AND NOT } B \{Q_2\} S) \Rightarrow P \{\textbf{if } B \textbf{ then } Q_1 \textbf{ else } Q_2 \textbf{ fi}\} R \text{ OR } S$$

If neither Q_1 nor Q_2 changes the value of B, we can replace the right-hand side of this inference by:

$$P \text{ AND } (B \text{ OR NOT } B) \{\textbf{if } B \textbf{ then } Q_1 \textbf{ else } Q_2 \textbf{ fi}\} \Rightarrow (R \text{ AND } B) \text{ OR } (S \text{ AND NOT } B)$$

I(2) **iteration**, for any I, B, Q and R:

$$I \text{ AND } B \{Q\} I \Rightarrow I \{\textbf{while } B \textbf{ do } Q \textbf{ od}\} I \text{ AND NOT } B$$

in which assertion I is called a loop-invariant.

Observe that the formulation of an invariant assertion for iteration structures can in fact only be used to 'test' the correctness of the loop body, not to 'prove' that it behaves as the programmer intended it to. Clearly, if the pre-condition of a block of code equals the post-condition, that block could (in so far as the proof is concerned) just as well be omitted from the program, which is most probably not the intention of the programmer. A way-out would then be to extend the iteration condition B in such a manner that (NOT B), will represent the desired outcome of the loop.

In the model introduced by **Nassi '74** the iteration structure can be associated with an inference rule without invariants. To this purpose Nassi defined control-flow structures with explicit exit or escape statements. In the consequent assertion of the iteration one can then distinquish between normal termination and termination via an 'escape-statement'. With the expression:

$$P \{Q\} (R,R')$$

one can then indicate that if P is true before the initiation of Q, and Q is terminated via an escape, then assertion R' must be true afterwards. If the execution of Q is not terminated via an escape, then assertion R must be true afterwards. The iteration structure can be described in this model as a repetitive block with one escape. The corresponding rule of inference is as follows:

$$I(2)' \; P \{Q\} (R,R') \Rightarrow P \{Q^\star\} (R',\text{false}),$$

where $Q^\star$ symbolizes the following iteration structure:

repeat Q; **if** R' **then break fi end**

Note that the execution of $Q^{\star}$ can only be terminated via the escape statement 'break'.

Remark:

Manna & Waldinger developed a method with which one can prove that a program is partially correct and will terminate, with one single proof procedure. They use 'intermittent assertions' of which one should prove not only that they are true whenever the point in the program text to which the intermittent assertion refers is reached, but also that this point will actually be reached at least once. For details we refer to **Manna & Waldinger '78, pg. 212-214.**

Though the choice of most intermediate assertions in the proof-procedure is more restricted in the Floyd/Hoare system, compared to Naur's method, it is still not a choice that can easily be automated or standarized.
The choice of the first antecedent (when working forwards) or the last consequent (when working backwards), and the choice of the loop-invariants, or loop-consequents (respectively, in Hoare's and Nassi's model), must be made with human ingenuity. The choice is seldom trivial. Too complex assertions can make the proof impossible, too simple assertions can make the proof useless.

Constructive approach **(Dijkstra '68)**
The constructive approach to program verification can be based on both the Floyd/Hoare model and the intuitive methods of Naur. The difference with these methods lies in the application.
Traditionally the verification problem is posed as follows:

> "Given a program and given a set of requirements, does the given program meet the given requirements?"
> **(Dijkstra '75, pg. 548).**

Dijkstra argued that programs are not 'given', as is implied by the above statement, but that programs are to be 'designed' first:

> "... by first choosing the proof pattern that **[the programmer]** wants to be applicable, he can then write his program in such a way as to satisfy the requirements of that proof."
> **(Dijkstra '75, pg. 548).**

The verification problem is then posed differently:

> "Given the specifications (-), how do we derive from these an algorithm meeting them ...?"
> **(Dijkstra '68, pg. 174).**

Similarly: the problem is also not to prove that a given procedure terminates, the problem is to design procedures such that (it *can* easily be proven that) they terminate.

The idea is then to start with a global description of the program's functions, annotated withglobal assertions, and then to refine both program *and* assertions stepwise until one reaches a description in a programming language, and one can formalize the implications between the refined intermediate assertions in first order logic.

Naur did describe these two approaches to the verification problem already in his article of 1966:

> "Our proof problem is one of relating a static description of a result to a dynamic description of a way to obtain the result. Basical-

ly there are two ways of bringing the two descriptions closer together, either we may try to make the static description more dynamic, or we may try to make the dynamic description more static."
(Naur '66).

More explicitly, Naur described the use of assertions in the design of loops:

"When writing (-) loops, general snapshots may be used as aids to construction, ..."
(Naur '66).

The program design method which is implied by the constructive type of program verification is known as 'structured programming'. Liskov summarized it as follows:

"The goal of structured programming is to produce program structures which are amenable to proofs of correctness. The proof of a structured program is broken down into proofs of the correctness of each of the components. Before a component is coded, a specification exists explaining its input and output and the function which it is supposed to perform."
(Liskov '72, pg. 193).

We will consider the structured programming methodology in more detail in chapter 3.

A more elaborate constructive proof method was described by **Wegbreit '77**. The method yields a program annotated with assertions *and* 'assertion-justifications':

"The justification establishes how the correctness proof is to proceed."
(Wegbreit '77, pg. 193).

For brevity we do not elaborate these methods further here.
For similar reasons we do not consider the theories for automated program synthesis in this context. (The interested reader is referred to **Manna & Waldinger '78, pg. 222-227.**)

Uninterpreted Program Schemas
Still, another approach to correctness analysis is based on among others the work of **Ianov '60**. (See **Elspas '72, pg. 98 - bottom.**)
In this field one studies abstract programs represented by uninterpreted program schemas. By considering the structure of the computations, instead of the contents one can try to derive properties that hold for entire classes of programs with equivalent structure.
One can study program equivalence and program transformations in this context **(Greibach '75)**.
Unfortunately, it was found that many interesting properties of program schemas are undecidable in general. **(Elspas '72, pg. 99 ; Greibach '75, parts VI-B and VI-C)**.

Remark:

Many of the graphical models which were discussed in section 2.2, can be considered as special abstract schemas for parallel programs. These schemas focus on interaction aspects, which implies that not only the program (control-flow) structures are formalized in the schema, but also that part of the code which is related to the enforcement of coordination rules. Ideally, these schemas are uninterpreted with respect to the *computations* performed, but interpreted with respect to the coordination

enforced. The relevant properties in a coordination analysis (deadlock, starvation, exclusion, reachability, etc.) are therefore not necessarily undecidable in such schemas.

With this brief outline of correctness analysis techniques for sequential programs we can do no justice to the research performed in this field. For brevity we have, for instance, omitted a discussion of proof techniques for recursive programs (see **Manna & Waldinger '78; Greibach '75**), and we could only briefly mention the problem to give proofs of termination **(Manna & Waldinger '78)**. For the purposes of this thesis, however, this outline suffices.

2.4.2. Analysis methods for parallel programs

If there are no interactions and no coordinations, the task of verifying a parallel program trivially reduces to the task of verifying a set of sequential programs, one by one. However, in this thesis we concentrate on processes which do interact, and which are coordinated, and in these cases the verification problem is more comprehensive. It is then wise to split the analysis in a number of subtasks. One can, for instance, first prove the observance of explicit coordination rules, and the absence of side-effects like deadlocks and starvations. One can then use the results of this first analysis in a proof of the functional correctness of the processes involved in the coordination. (See also section 2.1.) These distinctions are, however, not always made, as will appear below.

The analysis can be simplified if one can use special axioms and inference rules for higher level coordination primitives, like monitors and conditional critical regions. If no such higher level primitives were used in the system to be analyzed, one must appeal to simpler axioms on the indivisibility of, for instance, single read and write operations and the sequential character of the processes involved.

Gries '77 applied such a method to prove the correctness of a concurrent garbage collector. **Lamport '77** used it to analyze an exclusion algorithm. With these methods there is a tendency to leave the coordination requirements entirely implicit. In Lamport's view, for instance, a correctness analysis should consist of the following two parts: a proof of the correctness of each single sequential process, with the inductive assertion method, and a proof that the assertions on which the first proof is based are *monotonous* under (unaffected by) all possible executions of the competing processes.

Similarly, Dijkstra described how one can derive the *weakest antecedent* assertion for each critical section, from a section invariant which formalizes consistency properties of the interaction. **(Dijkstra '74**; cf. **Dijkstra '68)**.

Levitt '72 explored an analytical variant of this approach. ('Analytical', as opposed to 'synthetic' or 'constructive', implying that a given program is verified on given criteria.) Levitt, however, got entangled in complicated proofs, which led him to the frank conclusion:

> "Perhaps the main deficiency of the proofs presented here is that a suspicious reader might not believe the proofs."
> **(Levitt '72, pg. 44)**.

The proofs are indeed hard to check, and one would be tempted to ask for a 'proof of the correctness of the proofs'. (Note the freak theoretical danger of a 'recursive explosion of proofs', which would invalidate them all ...)

Hoare extended the inductive assertion method with proof rules for monitor-constructs **(Hoare '74)**. Hoare and Brinch Hansen have proposed to formalize the relevant properties of the data which are local to the monitor structure in *invariant assertions*, comparable to the loop-invariants discussed earlier. These invariant assertions must hold directly after the initialization of the monitor, before and after each monitor-procedure call, and before and after each execution of the WAIT and SIGNAL primitives.
The WAIT and SIGNAL operations require an extension of the set of axioms. We associate an assertion B with every condition variable b. The two additional axioms can then be formulated as follows:

A(2) I {b.WAIT} I AND B, and
A(3) I AND B {b.SIGNAL} I.

With these additional axioms one can analyze the functional correctness of single processes in a multiprocessing environment (see above), but one still cannot verify explicit coordination requirements, like absence of deadlocks and starvations. Oddly enough, some authors even found that these items need not be analyzed at all.
Dijkstra '68, for instance, argued that deadlocks are:

> "... more concerned with the problem as posed than with the task of programming it."
> **(Dijkstra '68, pg. 176)**.

In Hoare's view:

> "It is the responsibility of the programmer to avoid this risk [**of deadlock**]."
> **(Hoare '74, pg. 552)**.

Erroneously, however, Hoare concluded that the:

> "... assertion-oriented proof methods cannot prove absence of such risks, ..."
> **(ibid., pg. 552)**.

In 1976 Howard was able to devise a more powerful proof method for monitor structures by systematically adding explicit state variables to the procedure-bodies. Howard called these explicit variables *history variables*.
In fact, the idea of introducing additional variables to facilitate a proof is somewhat older.
Levitt '72 made use of such variables, which he described as:

> "... variables that we have introduced to simplify the extraction of the program's intent from the assertions."
> **(Levitt '72, pg. 36)**.

Owicki '75, '76 was the first to define proof rules in first order predicate calculus with which one can prove absence of deadlock and mutual exclusion properties. Owicki also uses additional variables in her proofs, which she calls *auxiliary variables*. The proof rules relate to concurrent processes which contain sequences of conditional critical regions. Critical sections for the same resource may not be nested. As discussed in chapter 1, the conditional critical region is in many respects equivalent to the monitor and it should not be difficult to apply Owicki's proof rules to monitor structures as well.

Instead of a single 'monitor invariant', Owicki defines a region invariant for each critical region in the system considered: I(r), where 'r' identifies the region. The region invariant I(r) must be true whenever no critical region for r is being executed.

Owicki defines auxiliary variables as follows:

> "Let AV be a set of variables of program S such that $x \in AV \Rightarrow x$ appears in S only in assignment statements of the form x := E where any variable may be used in E. Then AV is an *auxiliary variable* set for S."
> **(Owicki '76, pg. 282).**

She then introduces the 'auxiliary variable axiom', which we call A(4).

> *A(4) "If AV is an auxiliary variable set for S, let S' be obtained from S by deleting all assignments to variables in AV (and possibly some redundant* **begin end** *brackets). Then if P {S} R is true and P and R do not refer to any variables from AV, P {S'} R is also true."*
> **(ibid., pg. 282).**

(We have adapted the notation to the one used in this thesis.)
There are two additional theorems with which one can proof absence of deadlocks and mutual exclusions. We can summarize these theorems as follows:

T(1) Let S_1 and S_2 be blocks in different concurrent processes. The execution of the two blocks is assumed to be coordinated with the aid of conditional critical regions on variable 'r'. Neither S_1 nor S_2 may however itself belong to a critical region on 'r'.
Choose I_1 and I_2 such that for every statement S_1^i in S_1, and for every statement S_2^i in S_2 we have:

$$I_1 \; \{S_1^i\} \; I_1 \text{ and } I_2 \; \{S_2^i\} \; I_2.$$

(Assertion I_1 is invariantly true during the execution of S_1, and similarly assertion I_2 is invariantly true during the execution of S_2.)
Let I(r) be the invariant for 'r' (describing the 'reasonable states' of that part of the data base which is related to 'r').
Then if $(I_1 \text{ AND } I_2 \text{ AND } I(r)) \Rightarrow \text{false}$ (logical AND combination), the blocks S_1 and S_2 are mutually exclusive.

Observe that when S_1 and S_2, from the theorem above, are not mutually exclusive, there must at least be a state in which both I_1 *and* I_2 *and* I(r) are true, and thus the implication given in the last line of the T(1) would not hold.

T(2) Let C be a finite system of processes. Let Q_k denote the k-th process in C. Let Q_k^j denote the j-th critical region in process Q_k. Let B_k^j denote the condition in conditional critical region Q_k^j.
Let $P_k \; \{Q_k\} \; R_k$. Let $P_k^j \; \{Q_k^j\} \; R_k^j$. It is assumed that the condition in each conditional critical region is evaluated directly at the start of the region. Observe that this implies that region Q_k^j can only be initiated if $(P_k^j \text{ AND } B_k^j)$ is true.
Define D_1 and D_2 as follows:

$$D_1 = \bigcap_k \left[R_k \text{ OR } \left(\bigcup_j \left(\overline{B}_k^j \text{ AND } P_k^j \right) \right) \right]$$

$$D_2 = \bigcup_k \bigcup_j \left(\overline{B}_k^j \text{ AND } P_k^j \right).$$

If one can choose P_k, R_k, P_k^j, R_k^j and I(r) such that $(D_1 \text{ AND } D_2 \text{ AND } I(r)) \Rightarrow \text{false}$, then the system C is deadlock free.

Observe that D_1 formalizes a state in which no process is executing, either because the processes are completed (and thus R_k is true) or because they are

delayed in a region. D_2 formalizes a state in which at least one process is delayed in a region. If system C, from theorem T(2), contains a potential deadlock state, then both D_1 and D_2 will be true in at least that state, and then the implication given in the last line of T(2) would not hold. (More convincing proofs of these theorems can be found in **Owicki '75, '76.**)

Application

As a simple example of the application of theorem T(1) to a mutual exclusion problem, consider the following two concurrent programs[1]:

```
program 1: begin
                 region r wait (r2 = 0) do r1 := 1 end;
                 S1;
                 region r do r1 := 0 end
           end

program 2: begin
                 region r wait (r1 = 0) do r2 := 1 end;
                 S2;
                 region r do r2 := 0 end
           end
```

To prove that the execution of S_1 and S_2 is mutually exclusive, we introduce the following assertions. (Compare the symbols with those used in T(1).)

$$I(r) = (0 \leqslant r_1 + r_2 \leqslant 1),$$
$$I_1 = (r_1 = 1),$$
$$I_2 = (r_2 = 1).$$

Using the exclusion properties of critical regions as an additional axiom, it is not difficult to prove the truth of $I(r)$, I_1 and I_2. (We omit this proof for brevity here.) Evidently, (I_1 AND I_2 AND $I(r)$) $\Rightarrow$ false, which implies (with theorem T(1)) that S_1 and S_2 are indeed mutually exclusive.
To prove absence of deadlock between programs 1 and 2, we use T(2):

$$D_1 = (r_1 = 0 \cup (r_1 = 0 \text{ AND } (r_1 \neq 0 \cup r_2 \neq 0))) \text{ AND } (r_2 = 0 \cup (r_2 = 0 \text{ AND } (r_1 \neq 0 \cup r_2 \neq 0)))$$

We have taken the antecedent assertion of program 1 (P_1) equal to the antecedent of the first conditional critical region in program 1 (P_1^1), namely: $(r_1 = 0)$, and similarly for program 2, with $(r_2 = 0)$.
Observe that $D_1 \Rightarrow (r_1 = r_2 = 0)$.

$$D_2 = (r_1 = 0 \text{ AND } (r_1 \neq 0 \cup r_2 \neq 0)) \text{ OR } (r_2 = 0 \text{ AND } (r_1 \neq 0 \cup r_2 \neq 0))$$

Observe that $D_2 \Rightarrow (r_1 = 1 \text{ OR } r_2 = 1)$.
Clearly, D_1 AND $D_2 \Rightarrow$ false, and thus (with T(2)), it is proven that programs 1 and 2 cannot deadlock.

Observations

(1) The theorems introduced by Owicki give sufficient, but not always necessary pre-conditions for mutual exclusion and deadlocks. This implies that the proof rules may not be reversed to yield:

(I_1 AND I_2 AND $I(r) \not\Rightarrow$ false) implies that S_1 and S_2 are not exclusive,
(D_1 AND D_2 AND $I(r) \not\Rightarrow$ false) implies that system C contains a deadlock.

---

(1) The initial value of shared variables r_1 and r_2 is assumed to be 0.

(In variation to, respectively, theorem T(1) and T(2).)
Such theorems would be much more useful though. Observe that it may require much creative thinking and/or trial and errors to determine the right assertions I_1, I_2, I(r), etc., which yield the desired result via T(1) and T(2).

(2) The inductive assertion method, extended with the proof rules of Owicki and Howard does not allow us to verify coordination properties other than deadlock or simple exclusions:

> "There are many important correctness properties for parallel programs besides the ones treated here; priority assignments, progress for each process, blocking of some subsets of the processes in a program [**that is: starvation**] , etc. Many of these properties are difficult to define in a uniform way, while others require a language in which there are definite rules for scheduling competing processes."
> **(Owicki '76, pg. 285).**

These restrictions imply that (without further extensions) we cannot prove absence of starvation for a dining philosophers' problem, nor the obeyance of the precedence rules in the third readers & writers' problem, in this model.

(3) The importance of the use of explicated state variables in proofs, as demonstrated by, for instance, **Howard '76**, and **Owicki '75, '76**, implies that coordination primitives in which the manipulation of (sets of) such explicit state variables is foreseen and structured are at an advantage with respect to those in which such facilities are not present. The proofs become more comprehensive and thereby less convincing for the latter tools. As Owicki writes: the proof rules ...

> "... can also be modified to apply to programs which use other synchronization operations (e.g. semaphores, events) instead of [**conditional critical regions**] (-). However, the proof process becomes much longer in languages which do not restrict the use of shared variables."
> **(Owicki '76, pg. 284).**

For this reason we have based the taxonomy of coordination tools in chapter 1 on the extent to which the explicit use of state variables had been foreseen, and structured.

2.5. CONCLUSIONS

A. Graphical models

The graphical models which we have reviewed (Petri Nets, Coordination Nets, Slutz Graphs, Bigraphs, and Coordination Graphs) can be criticized on four major points with respect to their usefulness as analysis tools for coordination problems and their solutions.

(1) The models have insufficient power to model a wide variety of coordination problems in a transparent way. Only in an extended version of the Coordination Nets and in the Coordination Graphs could we represent (some) exclusion relations directly. In the other models it can be very difficult to distinguish computations from coordinations.

(2) The coordination relations that can be modelled by the analysis tools discussed, mostly do not yield conceptually satisfying representations of the problems involved. The exceptions are again the Coordination Graphs and the extended version of the Coordination Nets (see section 2.2.2).

(3) There are insufficient tools for the analysis of coordination problems with the aid of the models.

(4) There is often no clear correspondence between a graphical model and an implementation of the represented solution with standard coordination primitives. The exceptions are again the Coordination Graphs and the extended version of the Coordination Nets.

B. Formal specifications

(1) The formal specifications discussed here (Robinson & Holt specifications, Actor Model specifications, and Path Expressions) often lack conceptual clarity, especially for the more comprehensive coordination problems. One is not able to divide larger problems into combinations of smaller ones, for instance, by distinguishing systematically between exclusion effects and ordering effects.

(2) See point (3) above.

(3) See point (4) above.

C. Models based on predicate calculus

There are yet too few proof rules for analyzing a solution on all relevant types of coordination properties. There are no systematic procedures for finding the right assertions, invariants and verification conditions for a given solution. The modelled solutions lack descriptive clarity. (Observe that this is also not the purpose of these models. Their purpose is to provide the necessary analytical power to allow for the analysis of coordination problems.)

In the preceding sections we have studied to what extent some models obey the, relatively strong, criteria which one may pose to general analysis and design tools for coordination problems. An ideal model which would answer all requirements to the full will perhaps never be found, if only because some of the criteria are mutually conflicting (e.g. descriptive clarity and analytical power). Each of the models considered in this chapter has its strong points and its weaker points, but only by studying the weaker points in more detail one may hope to be able to improve them.

2.6. **REFERENCES**

Agerwala, T.K.M. (1975), *Towards a theory for the analysis and synthesis of systems exhibiting concurrency*, Ph.D. Thesis, John Hopkins University, 1975, 241 pgs.

Ashcroft, E.A. (1975), *Proving assertions about parallel programs*, Journal of Comput. Syst. Sci., Vol. 10, No. 1, 110-135, Febr. 1975.

Atkinson, R. & Hewitt, C. (1975), *Synchronization in actor systems*, Working paper, ACM SIGCOMM/SIGOPS Interprocess Comm. Workshop, Santa Monica, Cal., U.S.A., March '75, *Revised* Dec. '75, 13 pgs.

Belpaire, G. & Wilmotte, J.P. (1973), *See references chapter one.*

Belpaire, G. (1975), *On programming dependences between parallel processes*, Techn. Report, Comp. Sci. Dept., Univ. of Wisconsin, Madison, March 1975, Report No. 244, 61 pgs.

Bernstein, P.A. (1973), *See references chapter one.*

Boute, R.T. (1978), *Logical models for computer control of telephone exchanges*, Proc. Third Int. Conf. on Softw. Eng. for Telecomm. Switching Systems, Helsinki, June 1978, 18-24.

Brinch Hansen, P. (1973), *See references chapter one.*

Brinch Hansen, P. (1977), *See references chapter one.*

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, 89-102.

Cerf, V. (1972), *See references chapter one.*

Chen, R.C. (1975), *Representation of process synchronization*, Proc. of the ACM SIGCOMM/SIGOPS Interprocess Comm. Workshop, Santa Monica, Cal., U.S.A., March 1975, 37-42.

Dijkstra, E.W. (1968), *A constructive approach to the problem of program correctness*, BIT, Vol. 8, No. 3, 1968, 174-186.

Dijkstra, E.W. (1972), *The humble programmer - 1972 Turing Award Lecture*, Comm. ACM, Vol. 15, No. 10, Oct. 1972, 859-866.

Dijkstra, E.W. (1973), *A simple axiomatic basis for programming language constructs*, Techn. Report No. EWD 372, Univ. of Techn. Eindhoven, The Netherlands, May 1973, 18 pgs., Also in: Indagationes Mathematica, Vol. 36, 1974, 1-15.

Dijkstra, E.W. (1974), *Synchronisatie en sequencing (in Dutch)*, Informatie, Vol. 16, No. 12, Dec. 1974, 643-645.

Dijkstra, E.W. (1975), *Correctness concerns and, among other things, why they are resented*, Sigplan Notices, Vol. 10, No. 6, 1975, 546-550.

Dijkstra, E.W. (1976), *A discipline of programming*, Prentice Hall, Englewood Cliffs, N.J., 1976.

Elspas, B.; Levitt, K.N.; Waldinger, R.J. & Waksman, A. (1972), *An assessment of techniques for proving program correctness*, Computing Surveys, Vol. 4, No. 2, June 1972, 97-147.

Flon, L. & Habermann, A.N. (1976), *Towards the construction of verifiable software systems*, Sigplan Notices, Vol. 11, Special Issue, 1976, 141-148.

Floyd, R.W. (1967), *Assigning meanings to programs*, In: Proc. American Math. Soc., Symposium in Applied Math., Vol. 19, 1967, 19-32 (J.T. Schwartz (ed.)).

Gerhart, S.L. (1975), *Methods for teaching program verification*, SIGCSE Bulletin, Vol. 7, No. 1, 172-178. ACM/SIGCSE Fifth Techn. Symp. on Comp. Sci. Education, Washington, Febr. 1975.

Gostelow, K.P. (1971), *Flow of control, resource allocation, and the proper termination of programs*, Ph.D. Thesis, ENG -7179, Univ. of California, Los Angeles, Dept. of Comp. Sci., Dec. 1971.

Goodman, N. (1976), *Coordination of parallel processes in the actor model of computation*, Report no. TR-173, MIT, Dec. 1976, 116 pgs.

Gostelow, K.P.; Cerf, V.G.; Estrin, G. & Volansky, S. (1972), *Proper termina-*

tion of flow-of-control in programs involving concurrent processes, Proc. 1972, Annual ACM Conference, 742-754.

Greibach, S.A. (1975), *Theory of program structures: schemes, semantics, verification*, In: Lecture Notes in Computer Science, Vol. 36, G. Goos & J. Hartmanis (eds.), Springer-Verlag, 1975.

Greif, I. (1975), *See references chapter one.*

Greif, I. (1977), *A language for formal problem specification*, Comm. ACM, Vol. 20, No. 12, Dec. 1977, 931-935.

Gries, D. (1977), *An exercise in proving parallel programs correct*, Comm. ACM, Vol. 20, No. 12, Dec. 1977, 921-930.

Habermann, A.N. (1975), *Path expressions*, Report, Carnegie-Mellon Univ., Pittsburgh, Pa., June 1975, Comp. Sci. Department.

Hewitt, C.; Bishop, P. & Steiger, R. (1973), *A universal modular actor formalism for artificial intelligence*, Report IJCAI, Stanford Univ., Cal., Aug. 1973.

Hewitt, C. (1974), *Protection and synchronization in actor systems*, Art. Intell. Lab. Working paper No. 83, Mass. Inst. of Techn., Nov. 1974.

Hoare, C.A.R. (1969), *An axiomatic basis of computer programming*, Comm. ACM, Vol. 12, No. 10, Oct. 1969, 576-580, 583.

Hoare, C.A.R. (1974), *See references chapter one.*

Horning, J.J. & Randell, B. (1973), *See references chapter one.*

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

Howard, J.H. (1976), *See references chapter one.*

Ianov, Y.I. (1960), *The logical schemes of algorithms*, In: Problems of Cybernetics, Vol. 1, Translated from the Russian by Nadler, Griffiths, Kiss and Muir, Pergamon Press, New York, 1960, 82-140.

Karp, R.M. & Miller, R.E. (1967), *Parallel program schemata: a mathematical model for parallel computation*, IEEE Conf. Record of the eighth Annual Symp. on Switching and Automata Theory, Oct. 1967, 55-61.

Karp, R.M. & Miller, R.E. (1969), *Parallel program schemata*, Journal of Comp. and Syst. Sciences, Vol. 3, No. 4, May 1969, 167-195.

Keller, R.M. (1976), *Formal verification of parallel programs*, Comm. ACM, Vol. 19, No. 7, July 1976, 371-384.

Lamport, L. (1976), *Comments on 'A synchronization anomaly'*, Inf. Processing Letters, Vol. 4, No. 4, Jan. 1976, 88-89.

Lamport, L. (1977), *Proving the correctness of multiprocess programs*, IEEE Trans. on Softw. Eng., Vol. SE-3, No. 2, March 1977, 125-144.

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.

Levitt, K.N. (1972), *The application of program proving techniques to the verification of synchronization processes*, Proc. Fall Joint Comp. Conf., Anaheim, Cal., 1972, 33-47.

Liskov, B.H. (1972), *A design methodology for reliable software systems*, Proc. Fall Joint Comp. Conf., 1972, AFIPS Press, Montvale, N.J., 191-199.

Manna, Z. & Waldinger, R.J. (1978), *The logic of computer programming*, IEEE Trans. on Softw. Eng., Vol. SE-4, No. 3, May 1978, 199-229.

McCarthy, J. (1963), *Towards a mathematical science of computation*, Proc. IFIP Congress 1962, Münich, North-Holland Publ. Co., Amsterdam, 1963, 21-28.

Mills H.D. (1972), *Mathematical foundations of structured programming*, IBM, Federal Systems Division, Gaithersburg, Maryland, Report, Febr. 1972, FSC 72-6012, 62 pgs.

Nassi, I.R. & Akkoyunlu, E.A. (1974), *Verification techniques for a hierarchy of control structures*, Techn. Report No. 26, State Univ. of New York, Stony Brook, Dept. of Computer Science, Jan. 1974, 48 pgs.

Naur, P. (1966), *Proof of algorithms by general snapshots*, BIT, Vol. 6, 1966, 310-316.

Owicki, S.S. (1975), *Axiomatic proof techniques for parallel programs*, Ph.D. Thesis, Cornell University, 1975, 203 pgs.

Owicki, S.S. & Gries, D. (1976), *See references chapter one.*

Parnas, D.L. (1972), *On the criteria to be used in decomposing systems into modules*, Comm. ACM, Vol. 15, No. 12, Dec. 1972, 1053-1058.

Patil, S.S. (1970), *Coordination of asynchronous events*, Techn. Report No. TR-72, Mass. Inst. of Techn. Project MAC, June 1970, 240 pgs.

Petri, C.A. (1966), *Communication with automata*, Suppl. 1 to Techn. Report RADC-TR-65-377, Vol. 1, Griffis Air Force Base, New York, 1966, (Original in German: 'Kommunikation mit Automaten', Universität Bonn, 1962.)

Pratt, V.R. (1976), *Semantical considerations on Floyd/Hoare logic*, Report No. TR-168, Mass. Inst. of Techn., Lab. for Computer Sci., Sept. 1976, 40 pgs.

Robinson, L. & Holt, R.C. (1973), *Formal specifications for solutions to synchronization problems*, SRI Report, Stanford Research Inst., Computer Sci. Group, Menlo Park, Cal., Nov. 1973.

Robinson, L. (1975), *See references chapter one.*

Rodriguez, J.E. (1967), *A graph model for parallel computation*, Ph.D. Thesis, Mass. Inst. of Techn., Dept. of Electr. Eng., MAC-TR-64, Sept. 1967.

Sintzoff, M. & Van Lamsweerde, A. (1974), *Constructing correct and efficient concurrent programs*, MBLE Research Lab., Report R 266, Sept. 1974, Also in: Proc. Int. Conf. on Reliable Software, Los Angeles, April 1975, 319-326.

Slutz, D.R. (1968), *The flow graph schemata model for parallel computation*, Ph.D. Thesis, Mass. Inst. of Techn., MAC-TR-51, Sept. 1968.

Taub, A.H. (ed.) (1961), *Collected works of John von Neumann*, Vol. 5, Pergamon Press, New York, 1961, see 80-235: **H.H. Goldsteine & J. von Neumann**, *Planning and coding problems for an electronic computer instrument*, Part 2, Vol. 1-3 (see especially pg. 91; quoted in: **Elspas** *et al.* **1972, pg. 97**).

Tsichritzis, D.C. & Bernstein, P.A. (1974), *Operating systems*, Computer Science and Applied Math., W. Rheinboldt (ed.), Academic Press, New York and London, 1974.

Wegbreit, B. (1977), *Constructive methods in program verification*, IEEE Trans. on Softw. Eng., Vol. 3, No. 3, May 1977, 193-209.

Wirth, N. (1969), *See references chapter one.*

LIST OF ABBREVIATIONS USED IN CHAPTER TWO

(1) In readers & writers' problems

AR	active reader
AW	active writer
WR	waiting reader (reading request)
WW	waiting writer (writing request)

(2) In section 2.2.2

P	set of places in a Coordination Net
Ct	constraint set
R(Ct)	reduced constraint set
$R_a(Ct)$	subset of R(Ct) with only those constraints that contain place a
R(Ct(T))	reduced constraint set for transition T
I(a)	influence set of place a
EC(P)	set of equivalence classes of P
EC(Ct)	reduced constraint set of equivalence classes
IM(T)	set of illegal firing states (markings) of transition T

(3) In section 2.2.4

SESX	single entry single exit bigraph
PT	properly terminating
RF	repetition free
NL	necessarily looping
CFG	computation flow graph
TE	transition expression
LHS(t)	left-hand side of TE t
RHS(t)	right-hand side of TE t
$n_L(\sigma,t)$	number of instances of symbol σ in LHS(t)
$n_R(\sigma,t)$	number of instances of symbol σ in RHS(t)
$r = t(d)$	the result of the substitution of d in t is r
$t^{\overline{1}}$	initiation of vertex 1
$t^{\underline{1}}$	termination of vertex 1

(4) In section 2.2.5

R	relation of exclusions
C	relation of orderings

(5) In section 2.3.2

E	event
S	sequence of events
$\rightarrow$	direct continuation link
$\Rightarrow$	time ordering

(6) In section 2.4

S or Q	statement or block of statements
B	boolean condition
I	invariant assertion
P	antecedent assertion (pre-condition)
R	consequent assertion (post-condition)
AV	auxiliary variable set