SEMANTICS, VERIFICATION AND DESIGN OF CONCURRENT PROGRAMS USING ATOMIC ACTIONS

by

E. BEST

PhD Thesis

Submitted to the University of Newcastle upon Tyne

August 1981

Dedicated to the memory of

ATHANASSIOS KAPPOS

Acknowledgements

In the first place, I would like to extend my thanks to my supervisors, Peter Lauer and Brian Randell. Peter's and Brian's sympathy and readiness for discussions have guided me throughout the time I spent on this work and have greatly influenced its outcome. I am grateful for their comments on many drafts and "first versions" of this thesis. I am also happy to have been a member of their respective Research Projects during the five enjoyable years I spent in Newcastle. I am grateful for having had the opportunity to learn what I did learn during this time.

I would also like to thank my colleagues in these Projects who always had open minds for my problems and worries. Flaviu Cristian was instrumental in getting me to think about program specifications in the way described in section 2.5. Discussions with Graham Wood and Santosh Shrivastava have helped in finding the present form of chapters 3 and 4. Graham, in particular, drew my attention to the fact that the notion of an "immediate predecessor" needs to be modified for cyclic graphs (section 3.3.5); Santosh found a mistake in an earlier definition of a "contraction" (section 4.4). I also owe thanks to Pete Lee for critically reading drafts of chapters 3 and 4, and for giving me the right information at the right time, about an Operating System whose name shall remain unquoted. Section 6.4 is an outgrowth of many pleasant discussions I had with Fabio Panzieri about the algorithm described there. Finally, but by no means least, it is a pleasure for me to recall several enjoyable conversations with Mike Shields; sadly, it seems we didn't make as much of these as we possibly could have done. The influence of Mike's work, in particular his "firing sequence" formalism, on this thesis needs no further pointing out (section 5.2.2).

I am also indebted to Professor E.W. Dijkstra and Dr. M. Broy, discussions with whom (on the occasion of the 1981 Summer School on Functional Programming) helped me to appreciate the role of auxiliary variables in the Owicki-Gries proof method (section 5.2.5). The ideas on backtracking described in section 2.6 partly originate from discussions with Peter Henderson and one of his students, Simon Jones. My efforts in understanding the program described in section 6.2 were shared by P.J. Smith, a student in Newcastle. I owe thanks to John Rushby for kindly letting me use some of his many utility programs.

My warmest thanks are reserved for my wife, Monika, and my two little children, David and Robert, who regretfully all too often had to put up with an absent-minded husband/father. I sincerely appreciate all the support given to me by Monika; and David and Robert were foremost in my mind when I wrote the closing remarks of this thesis.

This work has been supported by the Science Research Council of Great Britain.

ABSTRACT

In this thesis we investigate the semantics and the design of concurrent programs using atomic actions.

On the semantic side, we define and compare two semantics for atomic actions: one which characterises atomic actions in terms of the executions they may give rise to and one which characterises atomic actions in terms of the effect relations associated with them.

We also give a relational semantics for "backtrack" programs which are claimed to be, in effect, simple concurrent programs.

We also study the semantic independence of programs.

On the design side, we present the design and proof of a few small but substantial concurrent programs.

CONTENTS

1. INTRODUCTION

1.1 Motivating Remarks

By a "concurrent program" I do not have in mind the opposite of a "sequential program". Rather, I tend to take the view that every (non-trivial) program contains elements of concurrency. A sequential program, for instance, usually involves two or more coexisting and at least partly independent variables. In a sequential program, concurrency is restricted in a special way, relating to the "flow of control". In my understanding, the adjective "sequential" merely refers to the "standard" order in which the commands contained in such a program can be executed; by "standard order" I mean the order in which the commands in question are arranged by control constructs such as concatenation, which is also usually the order in which they are executed on a sequential machine.

This standard order of executing a sequential program to achieve a desired effect does not necessarily have to be the only way in which the same effect can be achieved. For example, the standard way of executing the simple program P1.1 below would involve the setting of x to 0, followed by the setting of y to 1.

```
x:=0; y:=1
```

Program P1.1

However, if one is only interested in the end result established by P1.1, i.e. in the relation

$$x=0 \ \& \ y=1 \tag{1.1}$$

then a different order, say first setting y to 1 and then setting x to 0, would be equally acceptable.

Of course, there are programs whose order of execution really matters. For example, in

x:=x+1; x:=2*x

Program P1.2

the order of executing the two assignments is indeed vital; executing first x:=2*x and then x:=x+1 will give a different result, whatever the initial value of x. In general, the straightforward sequential execution of a sequential program will always establish the "strongest postcondition" which can be associated with that program; not necessarily, however, does there exist any other ordering with the same property.

If a programmer wishes to specify that two commands should occur in strict sequence then the "semicolon" sequencing operator, which is part of almost all programming languages, is an adequate tool. If, however, the programmer does not wish to specify any particular ordering between the commands he wishes to be executed then ordinary programming languages, as a rule, do not allow him to avoid doing so. In my understanding, the "parallel operator" || serves, in the first place, the purpose of allowing this sort of abstraction; i.e. it allows the programmer to group together a set of commands all of which he wishes to be executed in an unspecified order (or in no order at all).

There may be many reasons why a programmer should wish to make explicit use of this abstraction facility. An important one is efficiency. Any standard implementation of the above program P1.1, say, would first lead to the setting of x to 0 and then to the setting of y to 1. A more clever and efficient implementation can be conceived which somehow "detects" that these two commands are independent and translates them, perhaps on a "concurrent machine", into two concurrent (i.e. unordered) write accesses to x and y, respectively.

It can be imagined that the detection of such "independencies" between commands is a difficult task in general (we shall have more to say about this later). The programmer may therefore wish to indicate explicitly, by using the parallel operator, which commands could be

executed concurrently. For example, in

x:=0 || y:=1

Program P1.3

it does not require so much of a "clever" implementation in order to realise that the two assignments could be executed in parallel.

Whatever the reasons for using the parallel operator may be, I would like to stress my understanding of its important properties: it requires all of the commands it connects to be executed eventually, but leaves the order in which they can be executed unspecified. This latter property holds with some qualifications as to what should happen in case those commands are not independent. Indeed the main body of this thesis is concerned with the parallel combinations of commands which partly interact, and partly are independent. We discuss interaction shortly.

Thus, in this view the parallel operator is a means of abstraction (to abstract from irrelevant orderings), much similar to the usual non-deterministic operator which is also a means of abstraction (to abstract from irrelevant choices). The important difference between the parallel operator and the non-deterministic operator is that the former requires the execution of all commands it connects while the latter requires only the execution of one of the commands it connects. We shall find cause for discussing this distinction in great detail in this thesis.

If, in a parallel combination

$$c_1 \;||\; c_2, \tag{1.2}$$

the two commands c_1 and c_2 are independent then (1.2) is (by definition!) semantically equivalent to either $c_1;c_2$ or $c_2;c_1$ (independence ensures that the latter two are equivalent as well). If, however, c_1 and c_2 are not independent (let us say, through accessing a common variable) then (1.2) is, as yet, ill-defined.

Let us (for the sake of the argument) contemplate the definition of the parallel operator in such a way as to require the commands it connects to be independent. This would be attractive on more than one count. Firstly, its semantics then become trivial: we can just equate it with any one arbitrarily chosen sequential concatenation of the same commands. To all intents and purposes, the properties of the parallel combination of independent programs come out as the sum of their individual properties. Secondly, its implementation may become easier: we know then that there must exist a concurrent implementation.

However, I would not like to adopt this approach on the following two grounds. Firstly (and we shall go into this in more detail in this thesis) the notion of "independence" between commands is in itself surprisingly difficult to capture. The condition that the commands in question operate on different sets of variables is only a sufficient condition for the independence of these commands; it is however by no means necessary. We can see this immediately by comparing the two commands x:=y*0 and y:=1 which are independent but operate on a common variable. On the other hand, the condition that all possible sequential concatenations of the programs in question are equivalent (which is also sometimes called the "Church-Rosser" property) is only a consequence of the independence of these commands; it is by no means sufficient for them to be independent. Thus, we seem to need a fairly elaborate argument in order to characterise precisely this notion of independence.

Another (for me much more important) reason for allowing interacting programs to be combined by the parallel operator is that this is by far the more interesting, as well as practically useful, case. Very frequently one wishes to structure one's system into a set of commands, or "processes", which are independent "except for certain points of interaction". For the moment we can take "interaction" to mean simply the opposite of "independence". Depending on the form these interactions may take, we may distinguish various "models of interaction", and perhaps introduce (in addition to the parallel operator but to be used in connection with it) certain "primitives" which the programmer is free to use in order to effect the interaction he desires.

One such "model of interaction", which we will be particularly interested in, is what I call the "shared data model". This model, in its simplest form, is based on the assumption that all variables are "global" in the sense of being globally accessible by all parts of the program. It is evident that there must then be a rule governing the simultaneous accesses to those variables. Atomic actions, as I see their role, are in the first place a means by which the programmer can control and regulate such accesses. "Shared data programs" have frequently been considered, notably by Owicki and Gries in [88] and by Lamport in [67]; we outline the connections between our work and other related work in section 1.3 below. Throughout this thesis, I shall use the angular brackets < and > to enclose atomic actions; we shall consider an example in a moment.

Let me first stress that, with the above, I view atomic actions firmly as a notational tool which the programmer is free to use in order to express certain intentions. Atomic actions are therefore on a conceptual par with other programming tools such as, say, the conditional statement or, perhaps more fittingly, the Algol 60 block brackets begin-end. This is as opposed to a different possible view, whereby atomic actions are something "hardware-given" which the programmer is allowed to take account of, but is not allowed to redefine or otherwise interfere with. Our notion of an atomic action, emphatically, does not have any hardware connotation at all. The programmer thus being given the freedom in defining his own atomic actions as he needs them, we have to make sure that this freedom is exploited properly. For example: can we allow the nested use of atomic actions? This, and other related questions, will be answered in this thesis.

Let us now consider an example. Suppose that the action x:=0 in program P1.3 is to be implemented in the (admittedly stupid) way of loading the value of y into x and then subtracting y from x:

```
(x:=y; x:=x-y)  ||  y:=1
```

Program P1.4

In P1.4 there are now three explicit accesses to the variable y. Even granted that these three accesses are mutually excluded amongst each other, there is still the possibility that first x:=y occurs, followed by y:=1 and finally x:=x-y. In this case, assuming y has a well-defined value other than 1 initially, P1.4 behaves differently from P1.3.

If we want to make P1.4 and P1.3 equivalent, we need some means of expressing that the two assignments x:=y and x:=x-y "belong together". The atomic action brackets <> provide this means. By writing

<x:=y; x:=x-y> || <y:=1>

Program P1.5

the programmer is allowed to explicitly combine within an "atomic action" those pieces of program which he wants to "belong together". We postulate that, by definition of the semantics of the atomic action brackets, the program P1.5 is semantically equivalent to P1.3 (and thus also to P1.1). (This is, of course, under the assumption that y has a well-defined value initially.)

Programs using the parallel operator and atomic actions are the main subject of this thesis. Before focussing on such programs, I would like to point out briefly other instances of what I have called "models of interaction". We shall marginally discuss the notation called CSP (for "communicating sequential processes"), developed by Hoare [52] and others. In this model, variables are local to individual processes and cannot be accessed by others except through well-defined channels of communication. Communication occurs "handshake-wise", i.e. output and matching input are fully synchronised. Similar models are Lauer's COSY [73] and Milner's CCS [84].

In the above description, CSP seems to differ drastically from the shared data model outlined earlier. However, I think that as the formal semantics of CSP continues to be developed, we shall come to realise that programs written in CSP can be translated into programs written in the shared data notation, and vice versa. I believe, therefore, that it

will become a question of convenience rather than principle which notation can best be used to solve any given problem. We shall discuss the translation of CSP programs into shared data programs very briefly in this thesis.

We now concentrate on programs using, in addition to the more conventional sequential control structures, the parallel operator and the atomic action facility. The core of this thesis consists of formal definitions for a syntax, an associated semantics, and the correctness of such programs, Besides, we also discuss the design and proof of several small example programs.

In defining the formal semantics of such programs, we shall lay particular stress on the semantics of the atomic action brackets. Intuitively, atomic actions "occur instantaneously" (or "as single shots", as I once heard Dijkstra say in one of his lectures). These intuitive phrases did not, and do not, satisfy me because I felt it should be possible to find a more precise definition for the property they circumscribe. To find such a definition is one of the objectives of this thesis.

Frequently, one can find atomic actions being described by a sentence involving an "as if..." clause; such as: "atomic actions occur as if they did not take up time." Such descriptions often strike me as begging the question. Let us examine some of the literature for existing definitions of atomic actions. The first one I came across was the one by Lomet which can be found in [77]. I reproduce Lomet's definitions in full (using his numbering) because they illustrate the sorts of questions to be investigated in this thesis:

"The important properties of atomic actions can be expressed in a number of equivalent ways. We illustrate three.

1. An action is atomic if the process performing it is not aware of the existence of any other active process (can detect no spontaneous state change) and no other process is aware of the activity of this process (its state changes are concealed)

during the time the process is performing the action.

2. An action is atomic is the process performing it does not communicate with other processes while it is executing the action.
3. Actions are atomic if they can be considered, so far as other processes are concerned, to be indivisible and instantaneous, such that the effects on the system are as if they were interleaved as opposed to concurrent."

To my mind, these definitions raise more questions than they answer. For a start, it seems premature to call them "equivalent", since several key phrases used in them, such as "communication", have not been made precise. Granted that we know how to make these definitions precise, the statement that they are equivalent then seems to me a rather interesting theorem which I would have liked to see in full proof.

But let us examine (1)-(3) individually. As to (1), the phrase "being aware of the existence of ..." is so loose that I can associate almost nothing with it. The phrase "is aware" does not seem to be the same as the phrase "can detect"; why? And what about "spontaneous"? Presumably, (1) is intended to mean that other processes should refrain from "interfering" with the process in question while its atomic action is being executed. Understood in this way, (1) begins to make sense to me, and interference-freeness will indeed serve as one of our characterisations of atomicity.

Let us examine (2). By "communication" I always tended to understand communication in the "proper" sense, i.e. a process producing some value (of a variable) which is then read by another process. However, we shall presently show an example of a non-atomic action in which, in contradiction to (2), no such communication takes place. Consider

$$\langle x:=0;y:=0\rangle \;\|\; \langle y:=1;x:=1\rangle$$

Program P1.6

According to our understanding of atomic actions, this program will establish the final relation

$$(x=0 \ \& \ y=0) \ \vee \ (x=1 \ \& \ y=1) \qquad (1.3)$$

However, let us consider an execution of P1.6 in which first the two assignments x:=0 and y:=1 occur (perhaps concurrently) and then the two assignments y:=0 and x:=1. This establishes (x=1 & y=0), contradicting (1.3); yet no communication in the above sense has taken place.

It follows that we have to be very careful in defining what is meant by "communication"; it is not clear to me how this could easily be done. But let us turn to Lomet's third characterisation (3). The reference to indivisibility and instantaneity, of course, begs the question. The phrase that "the effects on the system are as if atomic actions were interleaved as opposed to concurrent" is however useful and can indeed, slightly rephrased, serve as an intuitive description of the atomicity criteria we will develop. A similar characterisation can be found in [34]: "We ... postulate that the net effect of our concurrently operating processes is as if atomic actions were mutually exclusive, i.e. the execution periods of atomic actions don't overlap."

The questions I associate with this definition are the following. Firstly, I am still unhappy with the rather informal "as if..." clause. Secondly, we are now defining the semantics of atomic actions, not in terms of their individual properties, but in terms of their surroundings; no longer do we have a definition of the form "an action is atomic if ..." but we have "actions are atomic if ..." or "processes using atomic actions behave as if ...". Can we not find a defining property associated with an atomic action all by itself? Thirdly, what is the above a property of? Is it purely a program which we are examining, i.e. are we defining the properties of atomic actions purely in terms of the program text? Or are we considering the set of executions of such a program, defining atomic actions in terms of these executions?

We may also consider Lamport's comments in [67]. He states that "atomic actions do not have internal control points". However, we can perfectly well imagine, say, program P1.6 to be executed in the following way:

x:=0; y:=0; y:=1; x:=1

which establishes (1.3) and gives rise to three well-defined intermediate states; we can, for example, without problem define the state "after y:=1" (which is an internal control point for the second atomic action in P1.6!) as (x,y)=(0,1). Perhaps it would be more to the point to say that "atomic actions behave as if they did not have internal control points". Here again, we have to elaborate on the phrase "as if ...".

Our approach to the definition of atomic actions will be based on a distinction between programs and their executions. We will first consider a single execution of a given program and define what it means for action executions to "occur atomically". This I call the "dynamic" characterisation of atomicity. Next we will consider a program as a whole and define what it means for one of its parts to be an "atomic action". This I call the "static" or "textual" characterisation of atomicity. We shall also derive a set of simple propositions linking dynamic and static atomicity criteria to each other.

Let me briefly describe the content of these criteria. Our dynamic criterion states that, in essence, atomic action executions must be interference-free with respect to each other. For the entire execution, this means that atomic action executions must form a partial order. Our static criterion states that atomic actions must be "effect-replaceable". That is to say, whenever an atomic action is replaced by a piece of program with the same "overall effect", then the semantics of the program in which the action is embedded does not change. This we shall have reason to consider as the characteristic property of atomic actions.

In formulating this latter criterion, we need to define precisely what is meant by "overall effect". To this end, we shall make use of what is known as "relational semantics". We shall work out an intimate connection between relational semantics and atomic actions. On the other hand, for our dynamic characterisation we need the notion of an "ordering" between atomic action executions. We capture this notion by making use of the so-called occurrence net model for the description of

executions. Both relational semantics and occurrence nets will be discussed in detail in this thesis.

Before giving a more detailed overview of the thesis, I wish to mention briefly some other issues which are also discussed in it. Returning to the parallel operator ||, I have said that this operator is similar to the non-deterministic operator (say, "OR") in the sense that both provide a means of abstraction (the former from ordering, the latter from choice). Their essential difference is that the || requires all commands it connects to be executed while the OR operator requires only one of the commands it connects to be executed. We will discuss this distinction in some detail. In particular, we shall argue that an operator which is known as the "backtracking choice" operator (see for example Cohen in [27]) is very akin to the parallel operator, rather than to OR. We shall define an operator called AND which behaves like the backtracking choice and is formally dual to OR. However we will not completely clear up the connections between AND and the usual ||.

The requirement that all commands connected by the || operator must eventually be executed gives rise to a special problem in connection with infinite loops. This is known as the "fairness problem" and we shall discuss our approach to it. Consider the following program, taken from [90].

$$\langle x:=1\rangle \;\;||\;\; \underline{\text{do}}\;\; \langle x=0\rangle \rightarrow \langle y:=y+1\rangle \;\; \underline{\text{od}}$$

Program P1.7

(In passing, we will also define precisely the semantics of an "atomic guard" such as <x=0> in P1.7.) Suppose (x,y)=(0,0) initially. The question is whether or not P1.7 should be required to terminate always, or whether it should be allowed to enter an infinite loop.

If we require P1.7 to terminate then some value of y must be a result, but it could be any value. The "weakest precondition" of such a program violates the "continuity property" accorded to weakest preconditions in [30]. This fact has been reason for some to reject this

approach, i.e. to allow the possibility that P1.7 does not terminate. P1.7 would loop forever if there were a "daemon" which would always unfairly choose the actions in the loop but would neglect the action <x:=1>.

My own position is, in line with Park's in [90], that P1.7 should be required to terminate. I would require the "daemon" (if we wish to think in such terms, which I do not particularly like because the "daemon" in question looks to me conspicuously like a euphemism for "sequential implementation") to be so fair as to choose <x:=1> eventually. I would postulate this as being in line with our requirement that the || operator leads to the execution of all of the commands it connects. If we allow the "daemon" to be unfair then we could hardly complain if in the case of

<x:=1> || <y:=1>

Program P1.8

(again with (x,y)=(0,0) initially) it chooses only to execute <y:=1> but refuses to execute <x:=1> and proclaims (x,y)=(0,1) to be a final state.

As an expense in "buying" fairness, I am quite willing to sacrifice the continuity of the wp, as I can find no cogent reason for insisting on it. I shall however also show, not only that fairness can be introduced in such a way, but that it can even be introduced fairly easily; all we need is a generalisation of a "maximality" property stating that all components of a concurrent program should be executed as far as possible. Thus, one loses the continuity of the wp, but one gains in exchange another "nice" property, namely that of maximality. I shall also show that by introducing fairness in concurrent commands one does not have to give up the usual [30] "unfair" interpretation of non-deterministic commands: the "daemon" which chooses between two simultaneously executable commands connected by the non-deterministic OR can continue to be as "erratic" as it likes (i.e. always choose one of them to the exclusion of the other).

I have said above that I consider there to be an essential difference between the parallel operator || and the non-deterministic OR. Nonetheless, the (rather imprecise) question has frequently been asked whether or not "concurrency can be explained by non-determinacy". That is to say: can we always translate a program using || into an equivalent program using only OR? Because the latter lead to continuous wp (unless some special statements are admitted) this is not possible: Pl.7, under the fair interpretation, cannot be translated into a guarded command program. However, "except for the infinite case" this translation is always possible, as will easily follow from our discussions.

1.2 Thesis Structure

There are two more or less introductory chapters, chapters 2 and 3. Two chapters (4 and 5) follow which, in the main, discuss the semantics of our programming language. Chapter 6, finally, contains three programming examples. Chapter 7 is the "obligatory" conclusion, but contains also some thoughts which I consider absolutely central to computer science (indeed to all science) and which worry me much more than the question of whether or not fairness should be adopted.

Chapter 2 describes relational semantics. We distinguish forward and backward semantics (of the latter, weakest precondition semantics are an instance). We investigate certain statements of equivalence between forward and backward semantics and we argue that the wp is somewhat restrictive in that it does not allow to distinguish between "possible non-termination" and "certain non-termination". We define a slightly more general semantics which we then continue to use in chapter 5.

Also in chapter 2 we develop an, in my opinion, rather attractive calculus for proving the correctness of programs, which distinguishes itself from the wp calculus in that it takes the initial values of variables into account as well. Thus, one becomes able to manipulate binary predicates in the same way as the more usual unary predicates. Further, we discuss non-determinacy. In particular, we discuss the OR operator,

and we define a dual AND operator which acts as a "disjoint split" operator. We show a connection between AND and backtrack programs.

In chapter 3 we introduce Petri nets in general and occurrence nets in particular. The latter will be used in order to describe executions of programs containing atomic actions. The atomic actions will act on the occurrence net as though "collapsing" parts of it; applied recursively, such collapsing gives rise to what we call "structured occurrence nets". We define a slight variant of the latter, which we call "structured occurrence graphs".

Chapter 3 also contains a short section on a property of infinite occurrence nets, which I believe to be connected to the continuity property of the wp mentioned earlier. Continuity of the wp has been postulated by Dijkstra in [30] with the intention, amongst others, of making it impossible to write a program which "makes within a finite time a choice amongst infinitely many possibilities". I shall argue that while there is reason to exclude sequential programs which "make a choice out of infinitely many possibilities", there is however no reason to extend this ban on concurrent programs.

In chapter 4 we define our dynamic atomicity criteria. We give a detailed motivation for our definitions in section 4.1. We aim at a definition as to when an execution of a given action, or piece of program, "occurs atomically".

We define a first criterion which, in essence, states that if the occurrence graph describing an execution is acyclic then it contains only atomic occurrences (section 4.2). We call this a "global" criterion because it is formulated as a property of an execution as a whole. Our second criterion (section 4.3) states that a portion of a given execution "occurs atomically" if it is not interfered with by its surroundings (in a sense to be made precise). This we call the "local" criterion. The correspondence between local and global criteria is also given in section 4.3.

Further, I attempt to relate these two criteria to what is known as the "two-phase lock protocol" [37]. I claim that the latter leads to

somewhat stronger forms of "atomic occurrences" which are context-independent (again in a sense to be made precise). We describe this connection in section 4.4.

Chapter 5 contains the formal definitions of our language: syntax in section 5.2.1, semantics in section 5.2.2, and correctness in section 5.2.4. Our syntax is rather general in that it allows the arbitrary nesting of concurrent programs within atomic actions, and involves a general scheme for combining atomic actions with other control constructs such as conditionals and loops. We define the semantics of our programs in the form of a relation between initial and final states.

In this definition, we use a mixture of "relational" and "operational" methods. We start with the idea that for the atomic actions contained in a program, all that matters as far as their environment is concerned is their effect relation. We therefore assume the effect relations of the atomic actions contained in the program to be given (perhaps calculated independently of each other) and we define the semantics of the whole program in terms of these effect relations and the way in which atomic actions are interconnected. The definitions of (partial and total) correctness then follow as a matter of course. In section 5.2.5 we discuss the connection between our semantics and the Owicki-Gries proof method.

By defining our semantics in section 5.2.2 we also implicitly define the semantics of the atomic actions contained in a concurrent program. This we call the "global static" atomicity criterion because atomic actions are defined only implicitly and because we derive their properties purely from the program text. In section 5.3 we work out a corresponding "local" criterion which, as discussed earlier, will be the property that one can replace an atomic action by any effect-equivalent piece of program without changing the semantics of the enclosing program. Section 5.3 is also the place where the connection between our various (four, to be precise) atomicity criteria is discussed.

In section 5.4 we turn to a question which has been mentioned previously, namely the "semantic independence" of two programs. We do no more than venturing an idea as to how this notion could best be captured

precisely. This idea involves the notion of variables being "transformed" into other variables so that the "semantic independence" of two actions becomes "syntactically visible". This is hoped to have a bearing on the determination of "maximally parallel" executions, but must largely be left to future research.

Finally, in chapter 6 we discuss the design and proof of three programming examples. The first example (section 6.2) concerns a fixpoint program first proved "in assertional style" by Dijkstra in [33]. We repeat his proof with a different derivation, and we add a proof conducted "in operational style". This is to show that our semantic framework can be used in a rigorous way. We also give a more detailed comparison between the two proofs.

Our second example (section 6.3) consists of a series of programs which compute an Euler cycle in a directed graph. Here we lay stress on the design process itself. In the proof of our last solution we use a combination of operational and assertional arguments.

In section 6.4 we present the design and proof of a fail-safe distributed extrema-finding algorithm, which is an extension of the Chang/Roberts algorithm described in [25]. Again, we lay stress on the design and on a fairly detailed proof. Also, in this section we make the more general point announced earlier, to the effect that CSP-type programs can be translated into shared data programs.

When planning the sections containing examples, I had hoped to be able to extract from these (and other) examples a number of design heuristics which could be collected together as possible general guidelines in the design of concurrent programs. I have, to my regret, not been able to do so, chiefly because the writing of everything else has taken such a long time that I could not get to the treatment of further examples. Unfortunately, therefore, the design aspects for the programs under consideration are less extensively investigated than their formal aspects. I hope, however, that future work will continue to provide insight into how concurrent programs can best be designed.

Chapters 3 and 4 of the thesis form a logical unit, referring to

the rest of the thesis only in an intuitive way. Chapter 5 is logically as much a sequel to chapter 2 as it is to chapter 4; for the understanding of sections 5.1 and 5.2, chapters 3 and 4 can be skipped. When the size of chapter 5 grew out of proportion, it occurred to me that it could without harm, and probably with benefit, have been split into two different parts, one (comprising section 5.2) located between chapters 2 and 3 and another one (comprising sections 5.3-5.5) following chapter 4. But it was then too late to make such a substantial change. I apologise to the reader for any inconvenience caused. Perhaps the short summaries can be of use which have been appended to every relevant subsection throughout chapters 2-5.

I should finally also mention that parts of this thesis have already appeared in print. In [9] some ideas are described which are also contained in section 6.2; however, section 6.2 is more detailed and contains also a different proof of the program in question. In [10] a program is described which occurs as an intermediate solution in section 6.3; however the final solution described in section 6.3 is new. The main ideas of chapters 3 and 4 have appeared in [11] and will also be contained in a forthcoming article [16]. However the motivation in chapters 3 and 4 (in particular section 4.1) has been extended; moreover [16] discusses error recovery, a topic which is not covered at all in this thesis.

1.3 History and Relation to Other Work

My interest in atomic actions stems from the work I have been doing in the two Research Projects I have been employed in. I started to work on path expressions [71,72] in 1976 while David Lomet was staying for a Sabbatical in Newcastle. David was at that time particularly interested in atomic actions and error recovery. He had proposed the idea [77] that a programming language should allow the programmer to declare, if he so wishes, any procedure he writes as atomic. When learning of this work, I thought it a good idea to use it in order to define a fully fledged concurrent language. I thought that one could benefit from replacing the "monitor" facility in Concurrent Pascal by a feature

whereby the entry procedures of a Concurrent Pascal process could be declared atomic, and another feature whereby the names of such processes could be grouped together into paths to achieve some desired synchronisation.

However, rather than pursuing this idea any further, I got interested in Lomet's three characterisations of atomic actions which are quoted in section 1.1 (originally, he even had four such properties). I felt that these characterisations left lot to be desired, or, in other words, to be found out. I thought that because atomic actions are intuitively so well-defined, it should be possible to find an agreed formal definition.

The second influence in my at first not very painstaking efforts aimed at such a definition came with a three-month-visit to Newcastle by Philip Merlin in 1977. He and Brian Randell came up with yet another idea of how to define atomic actions, described in [81]. Though I felt that their definition was very close to "the right thing" intuitively, I still found fault with it; this will be discussed in section 4.1 of the present thesis. In many ways, this thesis can be regarded as the result of my dissatisfaction with earlier definitions af atomicity.

In the meantime, it has been possible for me to establish several other connections, in particular that between atomic actions and relational semantics. Perhaps the outside work most closely related to the subject matter of this thesis is that of Owicki and Gries, described in [88], and that of Lamport, described in [67]. They consider the same sorts of programs, namely "shared data programs", as well as the same sorts of problems, namely formal semantics and correctness.

It is not possible for me to single out any one particular "result" or "new idea" contained in this thesis but not in such earlier work. I am quite sure that many people have previously had the same sorts of ideas as are described in this thesis. What I do claim, however, is that the semantic formalism developed in this thesis is altogether more comprehensive than similar previous approaches. In particular, we make formally precise several concepts which (to the best of my knowledge) have previously just been taken for granted without formal definitions,

such as the concept of an atomic action. In the remainder of this section I shall briefly describe the relationship to Owicki and Gries' work.

Let a concurrent program

$$c = c_1 \parallel c_2$$

be given. Owicki and Gries ask the question what properties can be deduced for c from the properties of c_1 and c_2. In the first instance, "property of c" means the "attachment of input/output assertions to c". In particular, in [88] they answer the question under what conditions the input/output assertions of c are just the logical conjunction of the respective input/output assertions of c_1 and c_2. This work thus yields, in the first place, a proof method for c.

We, on the other hand, ask ourselves the question what is the precise semantics of c. By (relational) semantics I mean, roughly, the strongest input/output statements that can be asserted about c. Our approach to defining this will be to take the relational semantics of the atomic actions contained in c_1 and c_2 (as opposed to taking the relational semantics of c_1 and c_2 as a whole!) and to define the semantics of c in terms of the way in which these atomic actions are grouped together in c_1 and c_2.

This approach is based on the idea that the atomic actions contained in c_1 and c_2 are just those parts whose relational semantics "really matters". One can put input/output assertions round an atomic action and study the behaviour of this action individually, knowing that (by their very properties) if an atomic action starts with an input assertion holding then it will end up with such an output assertion as can be calculated from its internal structure only.

Because Owicki and Gries propose to derive properties of c from input/output assertions around c_1 and c_2, and because c_1 and c_2 may be split up into several atomic actions, it may so happen that, using their method, certain statements about c cannot be proved without account being taken of the "internal control points" between these atomic

actions. In the Owicki/Gries method, it is necessary to introduce auxiliary variables for the purpose of expressing "internal control", and it may also be necessary to write down the strongest possible "intermediate" assertions, lest counterintuitive results be obtained. We shall discuss this in section 5.2.5.

Our approach differs from theirs chiefly in that internal control is taken into accout without the use of auxiliary variables. This is achieved by the use of "control sequences" which are defined in section 5.2. Control sequences describe internal control points "between" atomic actions, which (I will argue) are the only invariant properties of individual processes against outside interference, and as such must enter any semantic framework which claims to be appropriate. Internal control, to repeat, enters explicitly in our framework and implicitly in the Owicki/Gries approach.

Besides this difference in kind, there is also a difference in purpose. The Owicki/Gries theory is in the first place a proof method while our semantic formalism is in the first place a means for defining the semantics of the programs under consideration. However, there is also a proof method associated with our formalism, which will be illustrated on some examples in chapters 5 and 6.

I suppose that some people might call the proof method derived from our semantic formalism "operational" as opposed to the "assertional" method of Owicki/Gries. However, if any connotations of "implementation-dependency" or "informality" are associated with this term then I would reject it. Our formalism is just as "textual" as Owicki/Gries'. The difference is that the emphasis is shifted on how control can best enter the formalism. In the end, I think, both proof methods will come out as two sides of the same coin, or possibly two of the many sides of the "coin" under consideration. In every individual case, a decision has to be made as to which method can best be used. Thus, our work can perhaps also be seen as putting the Owicki/Gries method into perspective.

It should finally be mentioned that the term "atomic action" has also been used with a different connotation, referring to a program

which either achieves its desired effect or else "does nothing" (leaves the initial state intact) [68,28]. This "all-or-nothing" property seems to me a slightly extended property of correctness, which can be defined only with respect to the specification of a program and is therefore conceptually not related to our notion of atomicity. However there may well be practical connections in the sense that one may wish to design one's atomic actions in a concurrent program to have, in addition, the "all-or-nothing" effect. However, although the "all-or-nothing" property could easily be defined in terms of the concepts developed in section 2.5, we restrict our attention to atomic actions in the sense discussed informally throughout this introduction.

2. RELATIONAL SEMANTICS

2.1 Introductory Remarks

For the purposes of our discussion a semantic formalism is understood to be "relational" if it employs relations between the initial states and the final states of a program in order to characterise the semantics of the program. This is in contrast to other possible approaches (which loosely speaking may be called "operational") which would either take some intermediate states into account as well, or go as far as characterising a program by the "execution sequences" it generates.

Relational semantics in this general sense has a long history and has been so widely used that it is impossible to cite all references to it. Amongst the early papers in which the use of relational semantics has been suggested, perhaps the best known is [38] by Floyd. Hoare in [53] and Manna in [78] have advocated the reasoning about programs in terms of relations. Relational semantics has been further developed by Lauer in [70] and by Hoare and Lauer in [54]. Relational reasoning also pervades several important textbooks on program correctness, such as Manna's book [79], Dijkstra's well-known opus [30], and the more recent book by de Bakker [6].

The main reason for our present interest in relational semantics is that there is a natural connection between the relational formalism and atomic actions. This connection is to be explored in chapter 5 of this thesis. Besides, I also consider the present section to be interesting in its own right as a contribution to the relational semantics of non-deterministic and backtrack programs.

I propose to distinguish between two different forms of relational semantics, "forward semantics" and "backward semantics" (the same distinction has been made, for example, by King in [66] and by de Bakker in [6]). Forward relational semantics involves the association of a (set of) final state(s) to a given (set of) initial state(s), and vice versa for backward semantics.

The weakest precondition semantics [30] can, as we shall see (and as is well known), be viewed as a form of backward relational semantics. We shall describe a form of forward relational semantics in section 2.2 while weakest preconditions are described in section 2.3. These two semantic formalisms are "equivalent" in a sense also to be discussed in section 2.3. The reason for defining two essentially equivalent formalisms is that their practical application as calculi for the derivation of statements about programs is different. We shall also find (in section 2.6 and in chapter 5) that the forward semantics is, for our purposes, somewhat easier to manipulate.

The programs under consideration in this section may be non-deterministic. Non-determinacy is discussed in detail in section 2.4, with two aims in mind: firstly, to amplify some of the remarks made in section 1.1 about so-called bounded non-determinacy and continuity of the wp, and secondly, to prepare the ground for section 2.6 where we shall introduce a form of concurrency in a manner which is "dual" to the introduction of non-determinacy.

The semantic formalism expounded in sections 2.2-2.4 can be extended to so-called "binary predicates". This leads to a formal definition of a "specification" and of the "correctness" of a program with respect to a specification, formalising two concepts which are widely used in practice. This extension, described in section 2.5, forms the basis of a relational calculus for the total correctness of programs, which is illustrated on a few simple examples.

In our discussion throughout this chapter the "interpretation" aspects of the semantic formalism will be stressed. By this I mean all aspects of the formalism not relating to mathematical definitions and their consequences. In our final section 2.6 we discuss how the relational formalism, with very little changed mathematically, can be turned into a tool to describe a rather different (but nonetheless interesting) class of programs, simply by changing radically its interpretation. The class of programs in question is that of "pure backtrack programs" (again, a paper by Floyd [39] may be cited as one of the early references). Pure backtracking can, as it will turn out, be viewed as a

convenient method of implementing essentially concurrent programs, and a formal duality between concurrency and "ordinary" non-determinacy (of the kind discussed in section 2.4) can thus be established.

The programming language whose semantics is to be considered in this chapter is a slightly restricted form of the "guarded command" language of [30]. Our programs ("commands") have the following (hopefully self-explanatory) syntax, where capital letter words denote syntactic metavariables:

COMMAND ::= skip | abort | ASSIGN | IF | DO |
COMMAND;COMMAND

ASSIGN ::= VARIABLE := EXPRESSION

IF ::= if $BOOL_1$ -> $COMMAND_1$ []...[] $BOOL_n$ -> $COMMAND_n$ fi

DO ::= do BOOL -> COMMAND od.

The language defined in [30] allows in addition a "guarded list" to be included within the brackets do ... od. However via the equivalence

do $BOOL_1$ -> c_1
[] ...
.
.
.
[] $BOOL_n$ -> c_n
od

=

do ($\exists$j: $BOOL_j$) -> if $BOOL_1$ -> c_1
[] ...
.
.
.
[] $BOOL_n$ -> c_n
fi
od,

the more general case can easily be reduced to our less general case.

We assume that to every variable that can occur in a program there is associated a specific set of values which it may take. This set of values may in general be given by a "variable declaration", say as in Algol 60 or in Pascal. In the absence of such a declaration, we assume

a variable to range over the set of integers.

A "state" of a program is a mapping from the variables used in the program to their values. The set of all possible states is called the "state space" and is denoted by S. The state space itself can be viewed as a composite variable, its values being m-tuples of the values of its constituent variables. In other words, the state space can be viewed as the Cartesian product of the variables occurring in the program.

We assume (or make sure) that the evaluation of all expressions and assignments in a program always leads to values within the relevant value domains. If we drop this assumption then we enter the realm of "run time errors" and "exception handling". A discussion of this more general case is beyond the scope of this thesis, but extensive discussions are contained in [1,17,28].

2.2 Forward Relational Semantics

Our first aim is to associate to every program c a "relational meaning" or "effect relation" m(c) defined as

$$m(c) \subseteq S \times S \tag{2.1}$$

where the first S refers to the initial states and the second S refers to the final states of c. (This latter convention being the reason for calling m(c) a "forward semantics" of c.)

The relation m(c) is meant to capture the "input/output behaviour" of c. We adopt the following interpretation of m(c). The program c, when started in an initial state $s' \in S$, must terminate in some final state s such that $(s',s) \in m(c)$. If c may fail to terminate when started in s' then $s'm(c) = \emptyset$. This interpretation follows Wand's in [100]. We discuss it in detail in the sequel because it is important to be kept in mind that there are different possibilities for interpreting m(c). Failure to realise this may lead to confusion about the role of m(c).

We are at present interested in programs which are possibly non-deterministic. Let us define the general "prototype" of a non-deterministic command connective, denoted by "OR", as follows:

c OR c´ = if true -> c [] true -> c´ fi.

We imagine an execution of c OR c´ taking place by "non-deterministically choosing" between c and c´. If this choice is resolved in favour of c then c is executed and c´ is ignored; conversely, if c´ is chosen then c´ is executed and c is ignored.

Let us now see to what extent m(c) captures the input/output aspects of such a behaviour. In the simple example

skip OR (x := x+1)

Program P2.1

our interpretation of m(c) requires to include in m(P2.1) those pairs (s´,s) of states for which

$$s'(x) = s(x) \vee s'(x) = s(x)-1 \qquad (2.2)$$

This is because given an arbitrary initial state s´, P2.1 is indeed guaranteed to terminate in a final state s for which (2.2) holds. In this way, m(P2.1) captures the "overall behaviour" of P2.1.

Let us discuss our interpretation of m(c) on another example. We consider the program

skip OR abort

Program P2.2

To reiterate, our intuitive understanding of this program is that there is non-deterministic choice between "skip" and "abort"; if "skip" is chosen then P2.2 terminates properly, leaving the initial state intact,

and if "abort" is chosen, P2.2 does not terminate properly.

Because P2.2 may fail to terminate properly for every initial state s', our above interpretation of m(c) requires that in m(P2.2) the empty set $\emptyset$ be assigned to s'. In other words, knowledge of m(c) under its above interpretation does not allow us to distinguish between P2.2 and "abort". More generally, the above definition of m(c) leads to the two aspects "possible non-termination" and "certain non-termination" of the overall behaviour of a program to be treated equivalently.

For a comparison of this particular interpretation of m(c) with other possible ones we quote Harel in [49], p. 8:

> "... it is plausible to define the meaning of a program as a binary relation on states, including the pair (s',s) in that relation iff the program in question started in s' can indeed terminate in state s."

Because of the phrase "can ... terminate", it is clear that for the above program P2.2 the pairs (s',s') (but no other pairs) should be included in this (Harel's) relation. In other words, no distinction is now made between P2.2 and "skip". More generally, in Harel's interpretation, the two aspects "possible termination" and "certain termination" are not distinguished.

The point of this discussion is to show that there are indeed different possibilities of interpreting m(c), leading to quite different characterisations of the overall behaviour of a non-deterministic program. In fact, however, there are only two (namely the above two) possible interpretations of m(c) which reasonably reflect the non-deterministic behaviour intuitively pronounced above. This is because there is no other reasonable possibility than equating P2.2 either with "abort" (by letting the empty set $\emptyset$ correspond to every initial state) or to "skip" (by letting $\{s'\}$ correspond to every initial state s').

It follows that in order to distinguish properly between "skip", P2.2 and "abort", the simple definition (2.1) of m(c) is mathematically unsatisfactory: it does not leave enough room for interpretations other

than the above two. One way out, advocated frequently (for example, in [5,50,23]), would be to add to the state space a special element called "bottom" ($\underline{|}$) denoting the special "output state" in case the program fails to terminate.

With the help of such an element, programs could now be described by relations

$$m(c) \subseteq S \times (S \cup \{\underline{|}\}) \qquad (2.1')$$

which would enable "skip", "abort" and P2.2 to be distinguished [23]:

$$s'm(c) = \begin{cases} \{s'\} & \text{for } c = \text{skip} \\ \{\underline{|}\} & \text{for } c = \text{abort} \\ \{s', \underline{|}\} & \text{for } c = \text{P2.2.} \end{cases}$$

We shall find in chapter 5 the form (2.1') of the forward semantics more appropriate than (2.1) to capture the input/output properties of the concurrent programs we consider.

Despite its evident usefulness in (2.1'), I have always had misgivings about the "bottom" element, because just adding another element to the state space "feels" so arbitrary. Much rather than adding a special element to the state space I would prefer to be able to define a richer mathematical structure than (2.1), say relations over <u>subsets</u> of S, as relational semantics, thus admitting more room for interpretation. Such an approach seems possible and is sketched below at the end of section 2.6. It will turn out, amongst other things, that the addition á la (2.1') of a special element to the state space can be considered a consequence of a meaningful limiting property of such a "richer" semantics.

We return to a discussion of (2.1). We have seen that there are essentially two different possible interpretations of m(c). Without giving a value judgment, I claim that the first (i.e. Wand's and our) interpretation is the "forward counterpart" of weakest preconditions [30]. This claim is justified by the following two remarks. Firstly,

as will be seen in section 2.3, relations m(c) of the form (2.1) and weakest preconditions are interchangeble mathematical objects, i.e. a 1-1 correspondence can be found. Secondly, the interpretation of weakest preconditions agrees with ours: in weakest precondition semantics also, P2.2 is equated with "abort" rather than with "skip".

Because the second (Harel's) interpretation can apparently be formulated more smoothly than the first one, it tends to come to mind more easily and people may therefore be tempted to think that it is the only one. Because it does not however tie up with weakest preconditions, problems may arise if used in connection with it. Such is the case in [49] where Harel, in his chapter 5, has difficulties in squaring his interpretation with Dijkstra's and has to resort to different execution methods of non-deterministic programs in order to do so.

We now judge which one of the two interpretations is preferrable. There is in fact quite a clear-cut case. For correctness arguments we certainly cannot afford to treat a program which only possibly terminates, on an equal footing with a program which always terminates: this would be too "optimistic" and would leave room for possible unacceptable behaviour of the program under consideration. On the other hand, we are on the safe side if possible non-termination is treated as certain non-termination: this being a "pessimistic approach" would leave room for unexpectedly "good" behaviour.

To conclude this section, we give the relational semantics for our small language defined in section 2.1. (The reader is encouraged to check that P2.2 is indeed assigned the empty relation.)

$m(\text{skip}) = Id$

$m(\text{abort}) = \emptyset$

$s' m(x:=E) = \{s\}$, where $s(x)$=value of E in s'
and $s(y)=s'(y)$ for all variables $y \neq x$.

Define IF = $\underline{\text{if}}\ B_1 \rightarrow c_1\ []\ \ldots\ []\ B_n \rightarrow c_n\ \underline{\text{fi}}$,

where the B_i denote those subsets of S for which $BOOL_j$ is true;

$$(s',s) \in m(IF) \iff \forall j: (s' \in B_j \Rightarrow s'm(c_j) \neq \emptyset) \;\&\; \exists j: (s' \in B_j \;\&\; (s',s) \in m(c_j)).$$

$$(s',s) \in m(\underline{do}\ B \to c\ \underline{od}) \iff \exists k \geq 0,\ s_0,\ldots,s_k \text{ s.t.}$$
$$(i)\ s_0,\ldots,s_{k-1} \in B,\ s_k \notin B$$
$$(ii) \forall j \in \{0,\ldots,k-1\}: (s_j,s_{j+1}) \in m(c)$$
$$(iii)\ s'=s_0,\ s_k=s$$
$$\& \neg\ \exists\ s_0,s_1,\ldots:$$
$$(i) \forall_j \geq 0: s_j \in B \text{ and } (s_j,s_{j+1}) \in m(c)$$
$$(ii)\ s'=s_0$$
$$\& \neg\ \exists k \geq 0,\ s_0,\ldots,s_k:$$
$$(i)\ s_0,\ldots,s_k \in B$$
$$(ii) \forall_j \in \{0,\ldots,k-1\}: (s_j,s_{j+1}) \in m(c)$$
$$(iii)\ s'=s_0 \text{ and } s_k m(c)=\emptyset$$

$$(s',s) \in m(c_1;c_2) \iff s'm(c_1) \subseteq Dom(m(c_2)) \;\&\; (s',s) \in m(c_1) \circ m(c_2).$$

The last three formulae may seem unfamiliar (and it is hard work to prove the associativity of the semicolon ...!). The reason for them being unfamiliar is that they have been so defined that they correspond, under our interpretation, precisely to the better known and accepted weakest precondition definitions of [30] (see proposition 2.3 in section 2.3).

Notice, for example, that the definition of the concatenation operator "semicolon" does not just consist of the relational composition of its two constituent operations. There is also the requirement that the first operation leads into the domain of the second operation. This is because we wish to ensure consistency of interpretation, i.e. that from the meaning of the component operations a similar meaning for the whole operation is to be defined. In Harel's book [49], for example, the formula for the semicolon is different (it is just the relational composition); this is because of the difference in our respective interpretations of m(c), not because we are talking about different concatenation operators.

The fact that the forward counterpart of weakest preconditions does not give rise to "the natural" semantic definition of the semicolon (i.e. the relational composition) was also noted, rather as a matter of regret, by Hoare in [55]. In our discussion we have clarified why this is necessarily so, why this should surprise nobody, and we have also

seen that the "natural" definition of the semicolon can be restored when another interpretation of m(c) is considered.

Summary of section 2.2

We define forward relational semantics as a relation over the state space. This does not admit the three programs "skip", "abort" and "skip OR abort" to be properly distinguished. The latter program is equated either with "skip" or with "abort". The second possibility is taken in weakest preconditions. By introducing a special "bottom" element the three programs can be distinguished.

2.3 Predicate Transformers

Predicate transformers [30] are so called because they are functions whose domain and range are the set of output predicates ("postconditions") and the set of input predicates ("preconditions") of a program, respectively. A predicate Q is defined as a function

$$Q: S \longrightarrow \{true, false\} \qquad (2.3)$$

We also call such predicates "unary", to distinguish them from the more general objects to be defined in section 2.5.

Recall that the state space S can also be viewed as the Cartesian product over the variables of a program. Predicates are often represented as (first order) logical formulae involving these variables which evaluate to "true" or "false" for a given state, i.e. for a given assignation of values to variables. Predicates Q and subsets $X \subseteq S$ thus correspond uniquely to each other; this equivalence can be written as:

$$Q(s) = true \iff s \in X \qquad (2.4)$$

If (2.4) holds then Q is also called the "characteristic predicate" of X, and X is called the "truth domain" of Q.

The set 2^S of subsets of S forms a Boolean algebra [18] with the operations of set union and set intersection, and with the empty set $\emptyset \in 2^S$ and the full set $S \in 2^S$ as minimal and maximal elements, respectively. Via the equivalence (2.4), the set of predicates over S forms an isomorphic Boolean algebra with the corresponding operations of "logical or" and "logical and". Its minimal element (i.e. the characteristic predicate of $\emptyset$) is traditionally [30] denoted by F (for "identically false"); similarly, its maximal element is denoted by T (for "identically true"). Because of this isomorphism between subsets and predicates, we use the two interchangeably, in the sense that all definitions and statements about one can without problems be translated into corresponding definitions and statements about the other.

Let a program c and its effect relation m(c) be given. Two predicate transformers are of particular interest. We define for a subset $X \subseteq S$,

$$wlp(c,X) = \{s' \in S \mid s'm(c) \subseteq X\} \qquad (2.5a)$$

$$wp(c,X) = \{s' \in S \mid \emptyset \neq s'm(c) \subseteq X\} \qquad (2.5b)$$

Both the "weakest liberal precondition" wlp(c,.) and the "weakest precondition" wp(c,.) are thus functions from subsets of S into subsets of S, i.e. predicate transformers. As a consequence of the convention that m(c) relates initial states of c to final states of c, both wlp(c,.) and wp(c,.) relate (sets of) final states of c to (sets of) initial states of c, which justifies our calling them "backward semantics" of c.

Given our interpretation of m(c) in section 2.2, the formulae (2.5) determine the interpretations of wlp and wp. If X is a set of final states then (2.5a) indicates that wlp(c,X) contains all of those initial states s' such that <u>if</u> c terminates when started in s' <u>then</u> the final state will be a member of X. On the other hand, (2.5b) indicates that wp(c,X) contains all of those initial states s' such that c when started in s' is indeed <u>guaranteed</u> to terminate in a state in X.

The interpretation of m(c) was of course just so defined that these

interpretations of wlp and wp coincide with those given in [30]. I would like to stress that the other (i.e. Harel's) interpretation of m(c) would not have allowed m(c) and wlp(c,.) and wp(c,.) to be related by a simple relationship as in (2.5). Wand in [100] has in fact established that, in a certain sense, formula (2.5b) can be inverted. My next aim is to recall, and slightly generalise, Wand's result.

Using formula (2.5b) one notes that the following always holds:

Proposition 2.1 $wp(c,\emptyset) = \emptyset$ (2.6a)

$wp(c,X \cap Y) = wp(c,X) \cap wp(c,Y)$ (2.6b)

Let us call any function f from subsets of S into subsets of S (i.e. $f: 2^S \to 2^S$) "strict" iff

$$f(\emptyset) = \emptyset$$

and "multiplicative" iff

$$f(X \cap Y) = f(X) \cap f(Y) \quad \text{for all subsets } X,Y \subseteq S.$$

Proposition 2.1 then states that wp(c,.) is strict and multiplicative.

Wand's theorem states:

Proposition 2.2 Every strict and multiplicative function $f: 2^S \to 2^S$ is of the form (2.5b) for a suitable relation m(c).

Wand's result [100] is actually slightly less general than proposition 2.2, because he takes the continuity property (see section 2.4) into account as well.

Proposition 2.2 establishes a 1-1 correspondence between relations (2.1) and strict multiplicative functions from 2^S into 2^S. Since, as we have seen, the interpretations of m(c) and wp(c,.) are also compatible, it follows that the "forward semantics" m(c) (as defined by formula (2.1) and its interpretation) and the "backward semantics" wp(c,.) are

fully equivalent semantic descriptions of a (possibly non-deterministic) program c. It remains to be added that an analogue to proposition 2.2 cannot be found for weakest liberal preconditions.

Proof of proposition 2.2:

Let a strict multiplicative function $f: 2^S \to 2^S$ be given. Define a relation $m \subseteq S \times S$ by putting, for $s' \in S$:

$$s'm = \begin{cases} \emptyset & \text{if } s' \notin f(S) \\ \bigcap\{X \mid s' \in f(X)\} & \text{otherwise} \end{cases} \qquad (2.7)$$

It can be shown (by an argument similar to that employed in [100]) that $f(X) = \{s' \in S \mid \emptyset \neq s'm \subseteq X\}$.

To prove this for infinite state spaces one actually needs the infinite multiplicativity of f, i.e. the property that

$$f(\bigcap X_i) = \bigcap f(X_i)$$

for an infinite collection of sets $X_i \subseteq S$. Infinite multiplicativity of the wp does not seem to be provable from (2.6) only. We therefore just assume the property of infinite multiplicativity to hold. At any rate, we will not be interested too deeply in the infinite case, except for a few (marginal) discussions on the finite delay property and the property of unbounded non-determinism.

We end this section by recalling the wp definitions for our simple programming language [30].

wp(skip,X) = X

wp(abort,X) = $\emptyset$

wp(x:=E,Q) = Q[x←E],
where Q[x←E] is a copy of the predicate Q in which

all free occurrences of x are replaced by E.

$$wp(IF,Q) = (\exists j: B_j) \ \&\ (\forall j: B_j \Rightarrow wp(c_j,Q))$$

$$wp(\underline{do}\ B \rightarrow c\ \underline{od},Q) = \exists Q_i,$$
$$\text{where } Q_0 = \underline{not}\ B\ \&\ Q$$
$$Q_{i+1} = (B\ \&\ wp(c,Q_i)) \vee Q_0$$

$$wp(c_1;c_2,Q) = wp(c_1,wp(c_2,Q)).$$

These definitions are consistent with the definitions given in section 2.2; more precisely:

Proposition 2.3 (i) For the weakest precondition of a program c it is immaterial whether it is calculated directly by the formulae just given, or indirectly by calculating m(c) by the formulae given in section 2.2 and then transforming m(c) into wp(c,.) via (2.5b).

(ii) Conversely, m(c) is the same whether calculated directly or indirectly via (2.7).

The proof of proposition 2.3 is omitted.

Summary of section 2.3

Weakest preconditions with their two main properties, i.e. strictness and multiplicativity, and forward relational semantics (2.1) are equivalent semantic descriptions of a program. The formulae transforming one into the other are given in this section ((2.5b) and (2.7)).

2.4 Non-Determinacy

It is reasonable to call a program c "deterministic" iff

$$\forall s' \in S: |s'm(c)| \leq 1 \qquad (2.8)$$

In other words, c is "non-deterministic" if via m(c) two or more different possible final states are related to a single initial state: c must then halt in one of these states, but it is not determined a priori which one will be the actual final state.

This definition does not always accord with intuition. For instance, the program P2.2 = skip OR abort (section 2.2) turns out to be deterministic, contradicting the operational understanding. Similarly, the program

```
if x≤y → max := y
[] x≥y → max := x
fi
```

Program P2.3: Maximum

is also deterministic, even though if x=y initially then both branches of the conditional are eligible for execution.

The reason for this is quite simply that (2.8) characterises only such non-determinacy which can be asserted by inspecting the final state(s) after the program's assured termination. In P2.2, termination is not assured, and in P2.3 the two possible execution sequences do not lead to different final states.

The determinacy property can also be expressed in terms of weakest preconditions:

Proposition 2.4 m(c) is deterministic iff wp(c,.) is additive,

where a function $f: 2^S \to 2^S$ is called "additive" iff $f(X \cup Y) = f(X) \cup f(Y)$ holds for all subsets $X,Y \subseteq S$. Proposition 2.4 can be proved by observing the behaviour of formula (2.8) under the transformations (2.5b) and (2.7). Additivity of the wp is of course just the (intuitive) characterisation of determinacy given in [30].

The prototype non-deterministic command connective "OR", defined in section 2.2, can be expressed in terms of weakest preconditions:

$$wp(c \text{ OR } c',X) = wp(c,X) \,\&\, wp(c',X)$$

We return to this connective in section 2.6.

The notion of "bounded non-determinacy" [30] makes sense only for infinite state spaces S. It refers to the property that in m(c) only a finite number of final states correspond to any given initial state. We define a program c to be "of bounded non-determinacy" iff

$$\forall s' \in S: |s'm(c)| < \infty \qquad (2.9)$$

In fact, as shown in [30], the language defined in the previous sections always leads to programs of bounded non-determinacy.

In [30], chapter 9, two arguments are given as to why programs "ought to" be of bounded non-determinacy. Dijkstra's first argument refers to his (and our present) semantics of the loop construct; he notes that this semantics becomes intuitively invalid if programs of unbounded non-determinacy are admitted. In his second argument he reasons that a program of unbounded non-determinacy would be able to make "within a finite time a choice out of infinitely many possibilities" and would not therefore be implementable.

As noted in section 1.1, however, our position is indeed to allow concurrent programs of unbounded non-determinacy. To discuss the arguments for and against the axiom of bounded non-determinacy in more detail, we reconsider the following program [90]:

```
(x,y):=(0,0);
<x:=1>  ||  do <x=0 -> y:=y+1> od
```

Program P2.4

Because it is desirable to ensure that the left hand assignment of 1 to x will not be delayed indefinitely, i.e. that it will eventually be executed (see section 1.1), the program P2.4 must be guaranteed to terminate. But it is surely not possible to give an a priori bound on the

final value of y; this would introduce an intolerable element of arbitrariness. The effect relation m(P2.4) can thus be defined as containing, for any initial state, all of the infinitely many final states

$$(x,y) = (1,0),(1,1),(1,2), \ldots,$$

which violates (2.9). Our semantics which will be defined in chapter 5 gives exactly this result.

This example shows that the finite delay property for concurrent programs contradicts the axiom of bounded non-determinacy. This conclusion has also been reached by Chandra in [23] and by Park in [90]. Since we want the finite delay property to hold, it follows that the axiom of bounded non-determinacy ought to be dropped for concurrent programs in general, and it also follows that Dijkstra's two arguments in favour of this axiom must in some way be answered.

His first objection, that the semantics of the loop construct is too restrictive, can be overcome by finding an acceptable change in, or a generalisation of, the loop semantics. This line has been taken by Park in [90], Boom in [20] and by Back in [4]. Back arrives at the interesting conclusion that a loop semantics permitting unboundedly non-deterministic programs must of necessity be operational. I am not sure, however, whether this conclusion is really stringent; perhaps a "richer" relational semantics as briefly discussed in sections 2.2 and 2.6 will enable unbounded non-determinacy to be incorporated smoothly. At any rate, in chapter 5 we will define a loop semantics which does allow unbounded non-determinacy and therefore seems to be more general than the loop semantics defined in this section.

Dijkstra's second objection, that making a choice between infinitely many possibilities in a finite time is not implementable, is further discussed in section 3.4. The outcome of that discussion will be that his argument seems to make sense only if one has a sequential implementation in mind; for concurrent programs the argument in this form seems to be no longer valid.

Again, property (2.9) can be expressed in terms of weakest preconditions. A function $f: 2^S \to 2^S$ is called "continuous" iff for every ascending chain $X_1 \subseteq X_2 \subseteq \ldots$ of subsets of S, the equality $\bigcup f(X_j) = f(\bigcup X_k)$ holds. We then have:

Proposition 2.5 m(c) is of bounded non-determinacy iff wp(c,.) is continuous.

Continuity is of course just the characterisation of bounded non-determinacy given in [30]. Proposition 2.5 can be proved by using (2.5b) and (2.7). This result has also been noted previously, for example by Guerreiro in [46].

Summary of section 2.4

Non-determinacy (Nd) of the forward function characterises such Nd as can be asserted by inspecting final states after the program's assured termination. Nd corresponds to additivity of wp. Bounded Nd corresponds to continuity of wp; this axiom should be dropped for concurrent programs. Dijkstra's two arguments in its favour can both be overcome.

2.5 Correctness Criteria and Examples

In this section we describe a simple extension of the wp formalism. We consider the case in which the "goal" of a program is given as a relation between initial states and final states. Such relations occur frequently in practice as the specifications of the tasks to be performed by a program.

Our aim is to make precise this particular concept of a "formal specification" and then to define the "correctness" of a program with respect to its specification. We also illustrate some correctness proofs using the extended wp calculus.

Let us consider as an example the simple problem of writing a program to exchange the values of two variables x and y. More formally, we are required to find to every initial state s´ a final state s such that s(x)=s´(y) and s(y)=s´(x) (recall that states are functions from variables to values).

When unambiguous we follow the widely accepted notational convention of denoting the initial value of a variable x by "x´" (x primed) rather than s´(x) and its final value by "x" (x unprimed) rather than s(x)*. Our simple problem can then be stated as follows:

$$x = y' \ \& \ y = x' \tag{2.10}$$

Formulae such as (2.10), expressing a relation between the initial values and the final values of a number of variables, are frequently used as specifications of programs.

We consequently call such a formula a "specification" or a "goal"; formally we define a specification G as a "binary predicate"

$$G: S \times S \longrightarrow \{true, false\} \tag{2.11}$$

over the state space S where, as in (2.1), the first S stands for the initial states (expressed in G by primed quantities) and the second S stands for the final states (expressed in G by unprimed quantities).

Via the equivalence

$$G(s', s) = true \iff (s', s) \in m \tag{2.12}$$

* This convention must however be regarded as unsatisfactory, because it may lead to confusion if a variable and its values are denoted by the same symbol (which is like denoting a set and its elements by the same symbol). It would be preferrable in the long run to settle for a notation such as "preval(x)" and "postval(x)" to denote the initial and final values of x, respectively. Although I consider it important to find a solution to this notational problem, I would however not like to advocate any particular solution in this thesis.

there is again a natural isomorphism between binary predicates (2.11) and relations $m \subseteq S \times S$ which is analogous to the isomorphism (2.4) between unary predicates and subsets of S. As a result, we can henceforth use relations and binary predicates as interchangeable mathematical objects (where the emphasis is on "mathematical": by no means must we confuse the interpretation of m(c) given in section 2.2 and the interpretation of a specification given below!). All definitions about relations can therefore be transferred without problems to binary predicates, and vice versa.

In particular, a binary predicate G is called "deterministic" iff

$$\forall s' \in S: \mathrm{card}(\{s \in S \mid (s',s) \in G\}) \leq 1 \qquad (2.13)$$

Unary predicates (2.3) can be viewed as special cases of binary predicates, involving either only unprimed quantities ("postconditions") or only primed quantities ("preconditions").

Recall that in the wp formalism the function wp has been defined as a "predicate transformer" transforming a postcondition (which is a unary predicate) into a precondition (which is also a unary predicate). In this section we generalise the "wp" to a function which can have binary rather than unary predicates both as its arguments and as its results. It will be shown that binary predicates can be used quite analogously to unary predicates, and that this extension gives rise to a calculus of the "total correctness" of programs which is similar to that expounded by Manna in [79] in that it takes the initial values of variables into account in a satisfactory way.

Let us suppose that the specification G of a program is given as a binary predicate, as in (2.10):

$$G(\mathrm{Swap}) = (x = y') \;\&\; (y = x') \qquad (2.10)$$

We interpret this as imposing the following obligation on the implementor of G. If his program c is started in an initial state s' to which a final state s satisfying G exists (i.e. $s'G \neq \emptyset$) then c is required to

terminate in such a final state (due to non-determinacy of G there may be two or more such final states, any one of which is acceptable). If to s´ no such s can be found (i.e. s´G = ∅) then no particular behaviour is required of c, i.e. we don´t care what c does if started outside Dom(G)*. Notice that the domain of G(Swap) in our example (2.10) equals the entire state space, so that any implementation of G(Swap) is always required to terminate.

We have just given an informal account of what we would like to define as the "correctness" of an implementation c with respect to a specification G. To reiterate: within the domain of G, c is required to terminate in a state satisfying G. This definition is similar to the notion of "total correctness" defined in [79]. Formally, we call a program c a "(correct) implementation" of a specification G iff

$$\forall s' \in \mathrm{Dom}(G): \emptyset \; c \; s'm(c) \subseteq s'G \qquad (2.14)$$

Note that the definition (2.14) is weak in the sense that it allows a deterministic program c to implement a non-deterministic specification G. In the special case that any result of G could also be a result of its implementation c, it is plausible to call c a "precise implementation" of G. Formally,

$$\forall s' \in \mathrm{Dom}(G): s'm(c) = s'G \qquad (2.15)$$

It is in fact a very frequently occurring case that the specification of a program can most easily be stated as a non-deterministic binary predicate (stating: "we want a result but we don´t care which one out of a class of equivalent results"), while the implementation of this specification is deterministic, resolving the non-determinacy in the most convenient way; an example of this is given below. In fact, I consider the chief usefulness of non-determinacy to be its applicability at

* This last requirement can be strengthened. It is one of the chief objectives of "exception handling" to make programs well-behaved for all possible inputs by prescribing their behaviour outside the domain of their specifications [17,28]. The treatment of this question falls beyond the scope of this thesis.

the specification level where implementation detail can be disregarded.

We show as an example that the following program is a correct (and even a precise) implementation of (2.10):

```
x := x-y;
y := x+y;
x := y-x
```

Program P2.5: Swap

We will not in fact give the correctness proof of P2.5 by direct application of (2.14), but rather we will show how binary predicates can be used in a calculus which is rather similar to that of deriving preconditions by "backsubstitution" [30]. This method will be formally justified later.

Our method requires the specification G to be written at the end of the program c whose correctness with respect to G is to be checked. G is then backsubstituted through the program to its beginning. This backsubstitution follows exactly the same rules as the well-known method of backsubstituting unary predicates [53,30], except that the primed quantities of G (indicating initial values) are treated as constants; the substitution rules thus act only on unprimed quantities.

In each step of this backsubstitution, a new specification is obtained which may also contain a mixture of primed and unprimed quantities. The unprimed quantities indicate the "current states" while the primed quantities continue to indicate the initial states. After the last step, i.e. at the beginning of the program, initial states and current states can be identified, which can be done by "priming" all unprimed variables, thus transforming the originally binary predicate into a unary predicate.

We illustrate this method on our example:

```
1       {y=y' & x=x'}
2       x := x-y;
3       {y=y' & x+y=x'}
4       y := x+y;
5       {y-x=y' & y=x'}
6       x := y-x
7       {x=y' & y=x'}  =  G(Swap)
```

Program P2.5 (annotated)

Notice that line 5 has been obtained from line 7 by substituting the unprimed occurrence of x only, by the expression (y-x); similarly for lines 3 and 1.

We transform the binary predicate in line 1 into a unary predicate by "priming" all unprimed quantities, and obtain:

$\{y'=y' \ \& \ x'=x'\}$,

which is of course always true and hence the characteristic predicate of the state space S. From this, the correctness of P2.5 can immediately be inferred, because the domain of G(Swap) (see (2.10)) also equals S. The general rule is that c implements G iff the unary predicate obtained after backsubstituting and "priming" equals the domain of G; this rule is justified below in proposition 2.8.

We now proceed to justify this method formally. First we extend weakest preconditions such that they are functions from binary predicates to binary predicates. To this end, we consider the general situation that c is sequentially composed of c_1 and c_2 (see Figure 2.1); we are interested in the "weakest specification" for c_1 needed to ensure that c_2 will accomplish the goal G:

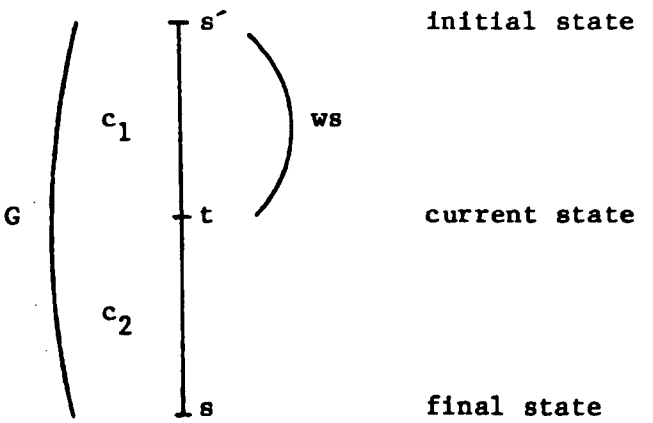


Figure 2.1

In this situation we define the weakest specification for c_1 which guarantees that G is implemented by $c = c_1;c_2$ as a relation

$$ws(c_2,G) \subseteq S \times S \quad \text{satisfying}$$

$$(s',t) \in ws(c_2,G) \iff \emptyset \; c \; tm(c_2) \subseteq s'G \tag{2.16}$$

An initial state s´ and an intermediate state t thus stand in relation $ws(c_2,G)$ iff c_2, when started in t, terminates in some final state s satisfying the global goal G. As an example, if c denotes the command in line k (=2,4,6) in the annotated version of program P2.5 and G denotes the specification in line k+1 then the specification in line k-1 equals ws(c,G).

From the considerations of section 2.3, especially when (2.16) is compared with formula (2.5b), it can immediately be appreciated that ws(c,.) is indeed a generalisation of wp(c,.). In general, we have for all specifications G:

Proposition 2.6 $wp(c,Cod(G)) = Cod(ws(c,G))$.

We have thus formally described the backsubstitution of binary predicates. It remains to describe the "priming" of a binary predicate. We have:

Proposition 2.7 When a binary predicate G $\subseteq$ S x S is "primed" then the characteristic predicate of the set $\{s \in S \mid (s,s) \in G\}$ is obtained.

Because apparently this set has some significance we coin a name for it: we define, for an arbitrary specification G,

$$Core(G) = \{s \in S \mid (s,s) \in G\} \qquad (2.17)$$

We can now state the justification for our correctness argument above as follows:

Proposition 2.8 A program c correctly implements a specification G iff Dom(G)=Core(ws(c,G)).

Proposition 2.8 follows immediately from the definitions (2.14), (2.16) and (2.17).

The intuitive meaning of the set Core(ws(c,G)) is that it contains all initial states for which c is indeed guaranteed to terminate in a final state satisfying G, even if c doesn't implement G as defined in (2.14); we therefore call the set Core(ws(c,G)) the "implementation domain" or the "standard input domain" of c with respect to G [17]. The standard input domain may in general be a proper subset of the domain of G. Proposition 2.8 states that if and only if these two domains are equal then c does indeed implement G.

Next in this section we give an example of a non-deterministic specification and a deterministic implementation, taken from [17,28]. Suppose that a program for the allocation of one out of N resources has to be written, $N \geq 1$. Suppose that the resources are represented by an array

var A: array (0..N-1) of {free,busy}.

The program should assign to a variable i the index of a "free" resource, and should turn the status of this resource into "busy".

The formal specification of this can be written as

$$G(\text{Allocate}) = (0 \leq i \leq N-1) \;\&\; (A'[i]=\text{free}) \;\&\; (A[i]=\text{busy}) \qquad (2.18)$$

Note that G(Allocate) is non-deterministic if more than one of the resources is free initially. The characteristic (unary) predicate of its domain is

$$\exists j: 0 \leq j \leq N-1 \;\&\; A'[j] = \text{free} \qquad (2.19)$$

We prove by the same method as previously that the deterministic program

```
i := 0;
do A[i]=busy → i := i+1 od;
A[i] := busy
```

Program P2.6: Allocate

correctly implements G(Allocate) as defined in (2.14).

Note that the behaviour of P2.6 is undefined outside the domain of G(Allocate); this does not however impede its being a correct implementation of G(Allocate) according to formula (2.14). In [17] a method is described supporting the insertion of tests which check for initial states outside the implementation domain.

The proof is conducted by annotating the program in the following, now hopefully self-explanatory, way:

```
1       {∃j: (0≤j≤N-1) & (A'[j]=free) & (A[j]=free}
2       i := 0;
3       {∃j: (i≤j≤N-1) & (A'[j]=free) & (A[j]=free}
4       do A[i]=busy → i := i+1 od;
5       {(0≤i≤N-1) & (A'[i]=free)}
6       A[i] := busy
7       {(0≤i≤N-1) & (A'[i]=free) & (A[i]=busy)}
```

Program P2.6 (annotated)

Again, by "priming" the binary predicate shown in line 1, the characteristic predicate (2.19) of the domain of G(Allocate) can be obtained, which by proposition 2.8 proves the correctness of P2.6.

In this section we have extended the wp calculus to binary predicates. We have shown that this yields a satisfactory calculus for proving the correctness of a program with respect to its (input/output) specification. In practice, of course, not all proofs will be as straightforward as the ones given, and the method outlined here should not be considered as a general recipe.

Note that only part of the equation of proposition 2.8 needs to be checked in order to prove correctness: because Core(ws(c,G)) $\subseteq$ Dom(G) always, only Dom(G) $\subseteq$ Core(ws(c,G)) needs to be proved. Such a proof may in practice be quite different in style from the two proofs given, using, for example, loop heuristics such as advanced in [63]. The points made in this section are that binary predicates can be used in such proofs (whatever their style) quite analogously to unary predicates, and that the "Core" operation plays a special role in proofs of the total correctness property (2.14).

The "Core" operation (i.e. the "priming" of a binary predicate) has also been defined in a recent paper [36] in which its above significance has however not been discussed. The question of how to incorporate initial values in the predicate calculus has also been tackled by Gries and

Levin in [45]; in my opinion their solution is less elegant than the calculus of binary predicates presented in this section, because they make a distinction between "real variables" and "logical variables" which seems unnecessary.

The reader may ask what the difference is between binary predicates and unary predicates using primed variables as "constant" symbols (or "logical variables"). My answer is that, mathematically speaking, no difference can be made. However, conceptually the use of binary predicates is in my opinion an improvement over using unary predicates with constants. Not only can "correctness" (2.14) be defined elegantly using binary predicates, but also other statements concerning the relation between the initial and the final values of variables can be formulated more elegantly; as an example, the reader is invited to rephrase the way in which initial values have been introduced in chapter 8 of [30] in terms of binary predicates. Furthermore and finally, whether binary predicates or unary predicates with constants are used, the "Core" operation must be defined similarly as above; it does seem to be easier to define it in terms of binary predicates.

Summary of section 2.5

The wp formalism is extended to "binary predicates" involving initial states and final states. A criterion for total correctness is defined and it is shown how the wp calculus can be extended to check this criterion. The extended calculus is illustrated on two examples.

2.6 Pure Backtrack Programs

In this section we consider a relational semantics which is in a certain sense "dual" to that defined in section 2.2. So far we have been concerned with the usefulness, and the limitations, of relations of the form (2.1) for the characterisation of the input/output behaviour of non-deterministic programs. By changing radically their interpretation, we can forge such relations to describe a different class of programs

which will here be called "pure backtrack programs" (for reasons to become clear later). Such programs have been studied extensively in the literature, for which the reader is referred to [27,19,39,104].

We aim at obtaining formal semantics and correctness criteria for pure backtrack programs. The ideas outlined here are not strictly used in later parts of the thesis, so that this section can be skipped. The reason for its inclusion is mainly its possible general interest. Besides, I shall argue that the programs considered here are effectively simple concurrent (rather than non-deterministic) programs, which allows us to classify them under the title of the thesis. However the connection to concurrent programs using atomic actions is not clear to me. We shall also present an argument by which the special "bottom" element used in (2.1´) can be derived, rather than has to be introduced ad hoc.

Wirth in [104] characterises backtrack programs as programs in which

> "... steps towards the total solution are attempted that may later be taken back when it is discovered that these steps lead into a `dead end street´".

Such programs often usefully arise in exhaustive search type problems. One particularly famous problem of this kind is the "8 Queens Problem" (see for example [39]) which, in its simplest form, asks for the enumeration of all possible ways in which eight queens can be positioned on a chessboard such that no queen attacks any other queen.

Backtracking techniques can usefully be employed in the solution of all kinds of other problems as well. However we shall concentrate at first on exhaustive search type problems, such as the 8 Queens Problem. These problems, in general, ask for the enumeration (or "production", as we shall say) of all of a set of "solutions". We use the adjective "pure" to indicate that backtracking is used in connection with such problems; the reason for this will become clear later.

To help in the concise formulation of a backtrack program the introduction of a special "choice" operator has been found convenient

[39,60,27]. This choice operator connects a list of "program branches". On its execution, one particular branch is chosen and executed until by means of some criterion this execution either "succeeds" (in which case its result is added to the list of "solutions") or "fails" (in which case it is just forgotten about). Upon establishing whether or not the execution of the most recent branch succeeds, the program then "backtracks" to the point where this branch had originally been chosen (i.e. the choice operator) and goes on selecting the next branch, until the list of branches is exhausted.

Because the order in which the program branches are selected may not be determined, programs using the "backtrack choice" operator just described have often been called "non-deterministic". I would however maintain that this kind of "non-determinacy" should not be confused with the non-determinacy which has been of interest in section 2.4. It is therefore very unfortunate that the same term has been used for two quite different things.

I wish to present my informal argument why the "backtrack choice" operator should not be called "non-deterministic". As described above, one of its characteristic features is that all of the program branches it prescribes have to be tried exhaustively for it to be validly executed. This is in stark contrast to the non-deterministic operator "OR" (see section 2.2) for whose valid execution it is sufficient that only one of its constituent commands be executed, to the exclusion of the other(s). Seen in this way, the backtrack "choice" operator behaves more like an "and" operator whose net effect it is to produce all of the solutions of the "successful" branches. The awareness of this distinction is not new and has been expressed, for example, in [21].

Because of this conjunctive property of the backtrack choice operator, a convenient way to think of its implementation would be the following. At the point of its execution the state space of the program is "split" into as many disjoint and independent copies as there are "branches" of the choice operator. Each one of these copies would serve as a starting point for executing the branch it corresponds to. If the execution of this branch succeeds, its final state is recorded as a

solution, and if it fails it is thrown away; in the end all of the solutions are collected together.

In effect, therefore, the backtrack choice operator could be viewed as a concurrent operator which "produces" all solutions simultaneously. Seen in this way, the above sequential description (i.e. that branches are chosen one at a time, separated by backtracking phases) can be viewed as but a sequential method (perhaps as the most natural such method) of achieving the same effect. The internal non-determinacy of the backtrack choice carries no significance with respect to its global effect: the important thing is that all solutions are produced, no matter in which order they are produced. That this kind of concurrency is a convenient way of looking at the backtrack choice operator has also been recognised early, perhaps most clearly in [60] where an Algol 60 extension incorporating backtrack choice is described and where the following quote can be found:

> "... `choice´ (which refers to our backtrack choice.EB) is not a magic function but a means of expressing parallel computation."

In sum, we may contrast the non-deterministic "disjunctive" choice operator "OR" (section 2.2) and the backtrack "conjunctive" choice operator whose net effect can be described as though all solutions were produced concurrently and which we therefore consider different from the OR. We call the latter operator "AND", in an obvious reference to its conjunctive properties. In this section we define the syntax and semantics of a simple language which incorporates the AND operator but no form of OR; i.e. all programs are "globally" deterministic but may contain concurrency in the form of AND (which the reader, if he so wishes, can think of as being implemented by backtracking). The programs written in this language will again be abbreviated by the letter "c".

Thus, we are interested in programs which produce, not one out of a set of equivalent results, but all results with a given property. The way to capture the semantic effect of such programs will actually consist, not in a change in the formula (2.1), but in a change in its interpretation. Previously, we have included (s',s) in $m(c)$ if c,

started in s', can produce s (give or take the problem of termination discussed in section 2.2). Now we include (s',s) in the input/output relation of c if c, started in s', is guaranteed to produce s. Thus, if, say, both (s',s_1) and (s',s_2) are members of the input/output relation then c produces both s_1 and s_2 from s'.

In order to minimise confusion we use a different letter for relations with our new interpretation. We characterise the input/output effect of a backtrack program c by a relation

$$b(c) \subseteq S \times S \tag{2.20}$$

over the state space, where again the first S refers to the initial states and the second S refers to the final states (and "b" stands for "backtracking").

If $(s',s) \in b(c)$ for an initial state s' and a final state s then we say that c "produces s from s'". The set of all final states produced by c from s' is $s'b(c)$. Thus, when started in an initial state s' then c produces all of the final states $s'b(c)$. In what way they are produced (one at a time with a label "success" attached to them or concurrently all at once) does not interest us.

In the special case in which $|s'b(c)| = \infty$ we may imagine that the production process be of infinite length and therefore non-terminating. One could here see an analogy to the case $s'm(c)=\emptyset$ for non-deterministic c. We do not however in this section wish to enter into a discussion of this case, and we therefore rule it out by making the overall assumption that S is finite.

In analogy to the considerations of section 2.5 we may also introduce "specifications" for backtrack programs, again as binary relations

$$H \subseteq S \times S \tag{2.21}$$

For $s' \in S$, we call a final state s a "solution" iff $(s',s) \in H$. We interpret H as requiring that an implementation c of H should produce all of the solutions specified by H.

We state the correctness of a backtrack program c with respect to a backtrack specification H in analogy to the formulae (2.14) and (2.15) given in the previous section. We call c a "(weak) implementation" of H iff

$$\forall s' \in S: s'H \subseteq s'b(c) \tag{2.22}$$

(2.22) requires that c produces all solutions but admits that c may produce some more than specified. One is in the case of backtrack programs probably more interested in programs which produce precisely the solutions specified by H:

$$\forall s' \in S: s'H = s'b(c) \tag{2.23}$$

We call such programs "(precise) implementations" of H.

We define our language first and give some examples later. Since for deterministic non-concurrent programs c our interpretations of m(c) and b(c) coincide, all deterministic constructs have the same semantics as before. The semantic formula b(c) for the AND operator is similar to the previous semantic formula m(c) for the OR operator. However, the interpretation of these formulae is different as discussed above. This is the reason for calling the programs defined here "dual" to those defined earlier.

skip: $s'b(skip) = \{s'\}$.

Intuitively, skip acts as expected: it produces s' from s'.

abort: $s'b(abort) = \emptyset$.

Thus, the program "abort" produces nothing out of s'; it acts as though "throwing away" s', and it could therefore be used to indicate the "failure" of a particular branch in backtracking.

x:=E: As before (we consider only terminating and deterministic assignments).

if B → c fi: $(s',s) \in b(\text{if } B \rightarrow c \text{ fi}) \iff s' \in B \;\&\; (s',s) \in b(c)$.

This "deterministic conditional" acts as abort if B is false and as c if B is true.

do B → c od: Same as before.

$c_1;c_2$: Same as before.

Define ANDCLAUSE = c_1 AND ... AND c_n:

This is where the production of possibly more than one solution comes in.

We define $(s',s) \in b(\text{ANDCLAUSE}) \iff \exists j: (s',s) \in b(c_j)$.

Thus, for the whole program ANDCLAUSE to produce a solution s, it suffices that one of the c_j produces s. The program ANDCLAUSE therefore produces all solutions that can be produced individually by the c_j. Note the similarity between this definition of AND and the previous definition of OR. Implicit in the definition of AND is that at its point of execution the entire state space is copied (not just the variables occurring in the c_j).

We now turn to a few examples. Let us consider the problem of producing all indices i in an array A[1..N] which point to negative array entries. We therefore have the specification

$$H(\text{Negentries}) = (A'[i] < 0) \qquad (2.24)$$

This could be implemented by the program

```
1        i:=1 AND ... AND i:=N;
2        if A[i]<0 → skip fi
```

Program P2.7: All Negative Entries

In line 1, N copies of the variable i are created, each pointing to a different array element. In line 2 of P2.7, all those indices are

collected for which the specification (2.24) holds; all others are thrown away. The program terminates with as many final states as there are negative entries in A[1..N]. Thus, P2.7 is a precise implementation of (2.24). The doubting reader might like to show, using the semantics of the language constructs just given, that indeed for every initial state s', the set s'b(P2.7) contains as many elements as there are negative array elements in s'.

As a more interesting example we reconsider the problem of the eight queens. Knowledge about this problem allows us to state a priori that no two queens can be placed on the same row. We are therefore asking to create, as many times as there are different solutions, an array Q[1..N] which contains for every row the column on which the queen of this row is placed. This specification assumes the form

$$H(8queens) = \forall i \in \{1,\ldots,8\}: safe(i,Q[i]) \qquad (2.25)$$

where "safe(i,j)" captures the condition that a queen on position j in row i is not attacked by any other queen on rows less than i:

$$safe(i,j) = \forall k \in \{1,\ldots,i-1\}: j \neq Q[k] \;\&\; |j-Q[k]| \neq |i-k| \qquad (2.26)$$

The following program solves the eight queens problem:

```
1       i:=1;
2       do i≤8 → j:=1 AND ... AND j:=8;
3                 if safe(i,j) → Q[i]:=j fi;
4                 i:=i+1
        od
```

Program P2.8: Eight Queens

Intuitively, the program works as follows. For every row (from i=1 to i=8), eight copies of j are created in line 2 to represent the eight possible positions on this row. (The arrays Q are also copied.) These copies are saved in the arrays Q iff a queen can safely be placed on that square (line 3), otherwise they are thrown away (together with

their corresponding Q's).

The reader can, if he so wishes, use the formal semantics of our language to show that, indeed, s'b(P2.8) contains all 92 solutions and nothing else, i.e. that P2.8 precisely implements H(8queens). We do not in this section give a proof of this fact. Note that the program can easily be extended to the NxN case.

Many combinatorial problems can be solved by programs of a similar type as P2.8. We may mention the "knight's tour", "nonequal adjacent sequences" and "binary circle" problems (see [43]), all of which can be solved easily and transparently by similar programs. A general form of such solutions is

```
"initialisation";
do "not exhausted" -> "branch_1" AND ... AND "branch_n";
                      if "partial solution" -> "record it" fi;
                      "next step"
od
```

Figure 2.2

This closely resembles the "general backtrack schemes" defined, for example, in [19] and [43]. Note however that the scheme shown in Figure 2.2 does not involve either recursion or implicit backtracking, in contrast to those other schemes. In fact, as argued above, backtracking can be seen as but a sequential method of implementing essentially concurrent programs, and I would further argue that recursion is but an elegant way of achieving backtracking. Note also that the use of AND is not limited to this case.

The ideas described in this section originate from an observation which came up in discussions between P. Henderson, S. Jones and the author, concerning the semantics of recovery blocks. I briefly describe this observation. Suppose that one wishes to define a program connective "AND" by forming the formal dual of the defining weakest precondition equation for "OR" (see section 2.4):

$$wp(c \text{ OR } c',X) = wp(c,X) \;\&\; wp(c',X) \qquad (2.27a)$$

$$wp(c \text{ AND } c',X) = wp(c,X) \vee wp(c',X) \qquad (2.27b)$$

Within the weakest precondition calculus, OR as defined in (2.27a) is a well-defined construct. However, AND as defined in (2.27b) is not well-defined. This can be inferred from the fact that formula (2.27b) does not preserve the property (2.6b), i.e. the multiplicativity, of weakest preconditions. That is to say, if wp(c,.) is multiplicative and wp(c',.) is multiplicative then it does not necessarily follow that wp(c AND c',.) as defined in (2.27b) is also multiplicative.

Nevertheless, if one pursues, in a purely formal way, the possibility of defining such an AND connective by means of (2.27b) then it turns out that such a connective must have properties similar to those ascribed to the "backtrack choice" operator. This is documented in [12]. Of course, when (2.27b) is seriously considered as a definition of AND then the wp framework is transcended. The question therefore arises whether or not this framework can be generalised to accommodate the AND operator.

I have tried to generalise the wp function for programs containing both OR and AND (assuming their semantics to be given intuitively as above), such that both (2.27a) and (2.27b) hold true, but have found this to be a rather intricate enterprise. I have therefore attempted at first to confine attention to programs using only AND but not OR, which I have here called "pure backtrack programs". The ideas contained in this section are the result of this limited approach. In particular, it has been easier to change the forward semantics rather than the backward semantics. The formulae (2.5), linking forward and backward semantics, can however also be redefined with b(c) in place of m(c) and given a correspondingly changed meaning; this is however beyond the scope of the present section.

I shall finally in this section outline how I would imagine the forward semantics of programs containing both AND and OR to look like. Such programs might be called "mixed" type programs, being in general neither purely non-deterministic nor pure backtrack (i.e. concurrent) programs. In practice there is likely to be some interest in mixed type

programs. For instance, suppose that one out of a set of results is to be produced but that the easiest way to produce this result is by first producing all results (say involving backtracking) and then selecting one of them; the solution could involve an AND program followed by an OR clause.

In general, we are now looking for a suitable forward semantics of a "mixed" program c such that the new semantics contains both m(c) (and its interpretation) and b(c) (and its interpretation) as special cases. An appropriate way to approach this objective seems to define a relation, which for obvious reasons I call "mb", over sets of states as follows:

$$mb(c) \subseteq S \times 2^S \tag{2.28}$$

The intention behind this definition is to include a pair (s',X) $(X \subseteq S)$ in mb(c) iff c, when started in s', can produce all of the states in X. Thus, non-determinacy arises in the sense that if $(s',X_1) \in mb(c)$ and $(s',X_2) \in mb(c)$ then it is not determined whether c will produce X_1 or X_2. Concurrency arises in the sense that any of the sets X which can be in $s'mb(c)$ can contain more than one element.

We show how m(c) is included as a special case in mb(c). If we impose on mb(c) the condition

$$\forall s' \in S: X \in s'mb(c) \Rightarrow |X| \leq 1 \tag{2.29}$$

then either only singleton sets or the empty set $\emptyset$ can correspond to an initial state s'. In this case mb(c) could equivalently be viewed as a relation

$$mb(c) \subseteq S \times (S \cup \{\emptyset\}), \tag{2.28'}$$

where the empty set $\emptyset$ now has the same function (and the same advantages) as the "bottom" element ($\perp$) discussed in section 2.2 (compare formula (2.28') with formula (2.1')).

(2.29) is a "no overall concurrency" or "no backtracking" condition. As such it may be related to a corresponding condition in [65], but (as this work has been done independently) I have not considered this possibility any further. However this possible connection is certainly to be looked at in further work.

On the other hand, b(c) is a special case of mb(c) in case the following condition holds:

$$\forall s' \in S: |s'mb(c)| \leq 1 \qquad (2.30)$$

In this case mb(c) could equivalently be viewed as a relation of the form (2.20), corresponding to an overall deterministic program c. In fact (2.30) is just a generalisation of (2.8).

These remarks give rise to hope that the forward relational semantics of "mixed" programs containing both AND and OR could suitably be defined by mathematical objects of the form (2.28). It is particularly satisfying that the addition of a special "bottom" element (see (2.1´) and (2.28´)) can be justified in a systematic, rather than an ad hoc, way as a consequence of the restriction (2.29). Note that objects of the form (2.28) are more general than weakest preconditions because, as seen in section 2.3, weakest preconditions are equivalent to a proper subset of $S \times 2^S$.

The generalisation considered in the form of (2.28) requires a corresponding generalisation (both intuitively and formally) of the basic language constructs. This is straightforward except for loops. To end this section, we give the generalised semantics for skip, abort, concatenation and the two operators AND and OR, and leave the loop semantics for future investigation.

$(s',X) \in mb(\text{skip})$ iff $X = \{s'\}$

$(s',X) \in mb(\text{abort})$ iff $X = \emptyset$

$(s',X) \in mb(c \text{ OR } c')$ iff $(s',X) \in mb(c) \vee (s',X) \in mb(c')$

$(s',X) \in mb(c \text{ AND } c')$ iff $X = X_1 \cup X_2$, where
$(s',X_1) \in mb(c)$ and $(s',X_2) \in mb(c')$

Note the formal similarity between the definitions of OR and AND: both are defined, essentially, by disjunctive formulae. However the interpretation of mb(c) ensures that OR corresponds to our non-deterministic connective while AND corresponds to the backtracking (or concurrent) connective.

$(s',X) \in mb(c_1;c_2)$ iff $\exists Y$: $(s',Y) \in mb(c_1)$ and $X = \bigcup\{X_y\}$
for some X_y such that $\forall y \in Y$: $(y,X_y) \in mb(c_2)$.

The above formulae for AND and OR can easily be generalised for conditional branching and splitting, respectively. As there are now two "conditional" constructs, it would seem plausible also to introduce two kinds of loop constructs. One kind of loop would correspond to the one used so far, e.g. in programs P2.6 and P2.8. The other would perhaps most conveniently act as though "producing" not only the very last final state after termination, but also all (or some) of the intermediate states which arise after full executions of the loop body. With such a loop, for instance, the assignments in line 1 of P2.7 and line 2 of P2.8 could be written more succinctly in loop form. However we defer the discussion of such an extension to future work.

Summary of section 2.6

By a change in interpretation, relations over the state space can be used to characterise backtrack programs. The change consists in switching from programs which produce one out of a set of results to programs which produce all results. Syntax and semantics of a simple backtrack language are given, illustrated by two examples. A possible extension to programs containing both non-determinacy and backtrack features is sketched.

3. PETRI NETS AND STRUCTURED OCCURRENCE GRAPHS

3.1 Introductory Remarks

Petri net theory (or, as we shall prefer to say, "net theory") is an outgrowth of a strand of reserach into concurrent systems which has been conducted since the early 60's [92,93,105,94,91,41]. We may quote [94] as to the general purpose of net theory:

> "The main purpose of net theory is to supply us with descriptive, deductive and conceptual devices:
>
> - descriptive devices for demonstrating the structure of systems; ...
> - deductive devices for solving application problems; ...
> - conceptual devices producing precise concepts on many levels, or promoting the communication between the computer expert and other people. ..."

The net formalism is based upon an underlying notion of "concurrency" which is defined as a symmetric (but not necessarily transitive) relation. At base, therefore, net theory is the study of symmetric relations [96].

A frequent way of applying nets is to describe the structure of a given system of interest by a net, while a "marking" of the net is used to represent the "current state" of the system (by indicating which system "elements" are "currently" holding). Changes in the system's state are modelled by a "transition rule" which governs the transformation of markings into one another. Nets used in this manner are usually called "system nets" [41]. A system net, its markings and the possible transformations of markings into one another are thus analogous, respectively, to a program, its states and the possible state sequences involved in an execution of the program.

As a consequence of the special form of the transition rule, every

possible marking transformation traces out of the system net a certain substructure of the net known as an "occurrence net". An occurrence net thus corresponds to one particular "execution" of the system net; as we shall see, it can be thought of as an "unwrapping" or "unfolding" of the original net. It so happens that the definition of a net can be cast in such general terms that both system nets and occurrence nets are "nets" in the sense of that definition (occurrence nets satisfying special properties).

The reason for our interest in nets is that in chapter 4 we intend to describe atomic actions in terms of the possible executions, or computations, they may give rise to. We explore the extent to which occurrence nets, in their capacity of corresponding to individual executions, are a suitable means for this purpose. The use of occurrence nets in this way has first been suggested by Merlin and Randell in [82]. It has been suggested in [82] as well as in [97] that the way in which atomic actions can best be represented is by the (possibly nested) "dynamic structure" they impose on the computation in question. To capture this idea, we use, in chapter 4, not plain occurrence nets but a more general concept called "structured occurrence nets".

In all, the present chapter contains a few general definitions and preliminary results in preparation of chapter 4. In section 3.2 we define nets in general, markings, the transition rule, and occurrence nets, and we motivate these definitions by an example. We further define structured occurrence nets in section 3.3; actually, for purposes to be explained in subsection 3.3.1, we define a slightly, and unsubstantially, modified version of occurrence nets which we call "occurrence graphs".

Section 3.4 is an "aside" or, more precisely, a "left over". It contains an argument, announced in section 2.4, to the effect that the axiom of bounded non-determinacy makes sense only for sequential programs, thus implying that it can be dropped for concurrent programs (that it should be dropped has already been argued in sections 1.1 and 2.4). The reader can omit section 3.4 without any loss of continuity. In order to understand the main ideas described in chapter 5, the reader

can also omit chapters 3 and 4 altogether.

3.2 Nets and Markings

This section contains the relevant definitions from [42] in insignificantly modified form. A "net" is defined as a triple (S,T,F) where

$$\begin{aligned} &S \text{ is a nonempty set of "state elements" or "places"} \\ &T \text{ is a nonempty set of "transitions"} \quad (S \cap T = \emptyset) \\ &F \subseteq S \times T \cup T \times S \text{ is the "flow relation".} \end{aligned} \tag{3.1}$$

If (as is common practice [41]) places are represented by circles and transitions are represented by boxes then it is convenient (and usual) to represent F by arrows between boxes and circles. The definition of F requires that an arrow belonging to F may only lead from circles to boxes or vice versa (never from a circle to a circle or from a box to a box). Examples may be found below.

The role of a place is to model any aspects pertaining to the "states" of a system. In a program, for instance, the possible values of a variable could be represented by a set of places, as shown below on an example. Transitions, on the other hand, serve to model any aspects relating to state changes, such as the "commands" of a program.

This duality between state elements and transitions I consider to be one of the distinguishing and most useful aspects of net theory. The subject matter of duality can be, and has been, developed much further; the reader is asked to consult [74,75,56,40,80]. In case no distinction is to be made between states and transitions we use the neutral term "element" and define the set of all elements as $X = S \cup T$.

Notationally, there does not, unfortunately, seem to be any easy way around using the same symbol S for the set of places in a net and for the set of states in a program (chapter 2). However no confusion will arise from this, because we shall, except in this section, not use the (S,T,F) form of nets. These two sets, as seen above, may in any case play quite similar roles.

To end the definition of nets we note that nets can, mathematically speaking, be considered as directed bichromatic graphs [8] whose nodes are the elements of S and T, respectively, and whose edges are the elements of F. All of the usual graph theoretic definitions such as strong connectedness and cycle-freeness can therefore be transferred without problems to nets as well. When referring to nets, we shall always use such definitions in agreement with established terminology.

A "marking" of a net (S,T,F) is defined as a function

$$M: S \longrightarrow \mathbb{N}_0 \tag{3.2}$$

where $\mathbb{N}_0$ is the set of non-negative integers. A marking can be represented pictorially by placing as many "tokens" on a place s as its marking M(s) indicates. If two places are both marked in a marking (i.e. their token content is greater than 0) then they are said to "hold concurrently".

In a marked net a transition $t \in T$ is called "enabled" iff

$M(s)>0$ for all $s \in Ft$.

An enabled transition can "occur" or "fire", thus (by definition) transforming the marking M into a new marking $\bar{M}$ defined by

$$\bar{M}(s) = \begin{cases} M(s)-1 & \text{if } s \in Ft \setminus tF \\ M(s)+1 & \text{if } s \in tF \setminus Ft \\ M(s) & \text{otherwise} \end{cases} \tag{3.3}$$

We illustrate this transition rule by the following example:

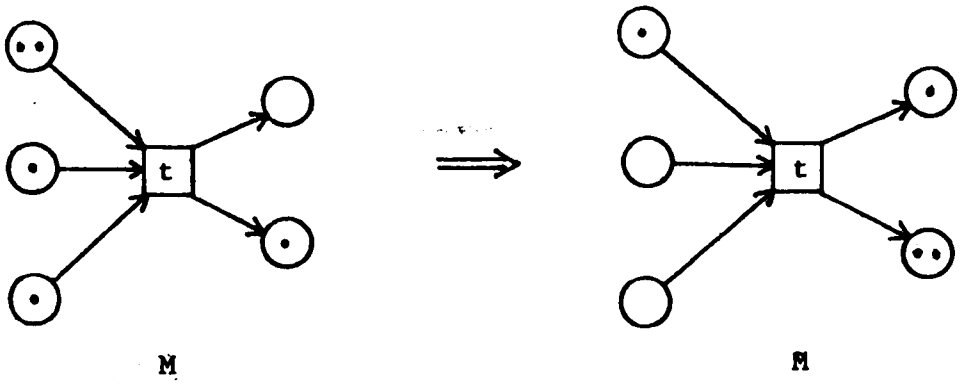


Figure 3.1

A marking M̃ is called a "successor marking" of M if there is a sequence of transition firings transforming M into M̃. For system nets, a marking M and the set of its successor markings [61,48] form the possible "behaviour" of the system, given that M is its initial marking. Not very surprisingly, the set of successor markings of M is usually enormously complicated (by its sheer size alone). It is therefore one of the chief objectives of net theory to discover statements which allow one to deduce from properties of the system net itself (the "static structure") certain properties of the marking graph (the "dynamic structure"). Most of the work relating to nets involves such statements.

In this section we shall be interested in the links between static structure and dynamic structure only to the extent of pointing out on an example that the definition of an "occurrence net", being a formalisation of the notion of markings being transformed into one another, follows from the transition rule. We give the example first and define occurrence nets afterwards.

Consider the following net:

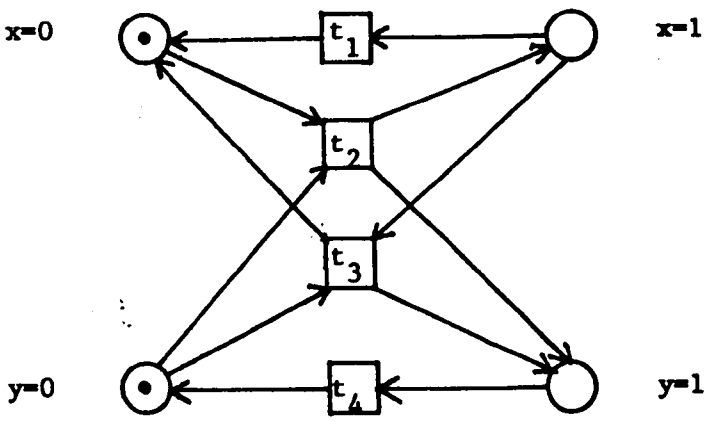


Figure 3.2

With the inscriptions shown, this net can serve as a representation of the following program:

```
var x,y: 0..1;

(x,y) := (0,0);                          % set up initial marking
do x=1 -> x := 0                         % t1
[] y=1 -> y := 0                         % t4
[] (x,y)=(0,0) -> (x,y) := (1,1) % t2
[] (x,y)=(1,0) -> (x,y) := (0,1) % t3
od.
```

Program P3.1

The net shown in Figure 3.2 and program P3.1 correspond to each other in that every firing sequence in the net corresponds to a valid execution of the program, and vice versa. Let us consider the state reached after executing t_2 and t_4; the state is then (x,y)=(1,0). The next occurrence from this state could either be the occurrence of t_1 or the occurrence of t_3, but not both! In other words, both the net and

the program contain proper non-determinacy.

In state (x,y)=(1,1), on the other hand (reached after firing t_2 only), t_1 can occur concurrently with t_4, a possibility which is explicit in the net but only implicit in the program. This distinction is due to the fact that the concurrency implicitly present in the program in the form of two variables (compare section 1.1) manifests itself in the net by the explicit presence of two tokens.

When the firing of transitions in a particular firing sequence (i.e. a single "execution" of the net) is accounted for by keeping track of the actual "movement" of tokens, then this can be represented by an "unwrapping" of the net which connects the transitions and the places of the net in the order in which they actually occurred or held, respectively.

As the net may be repetitive, such an "unwrapping" may contain several distinct occurrences of the same transition, as well as several distinct holdings of the same place. We call any single occurrence of a transition an "event" and any single holding of a place a "condition" [41].

If events and conditions are again represented as boxes and circles, respectively, then the "unwrapping" itself can be represented as a net. The following Figure shows an example of such a "single execution" in which the actual occurrences have been distinguished by running upper indices:

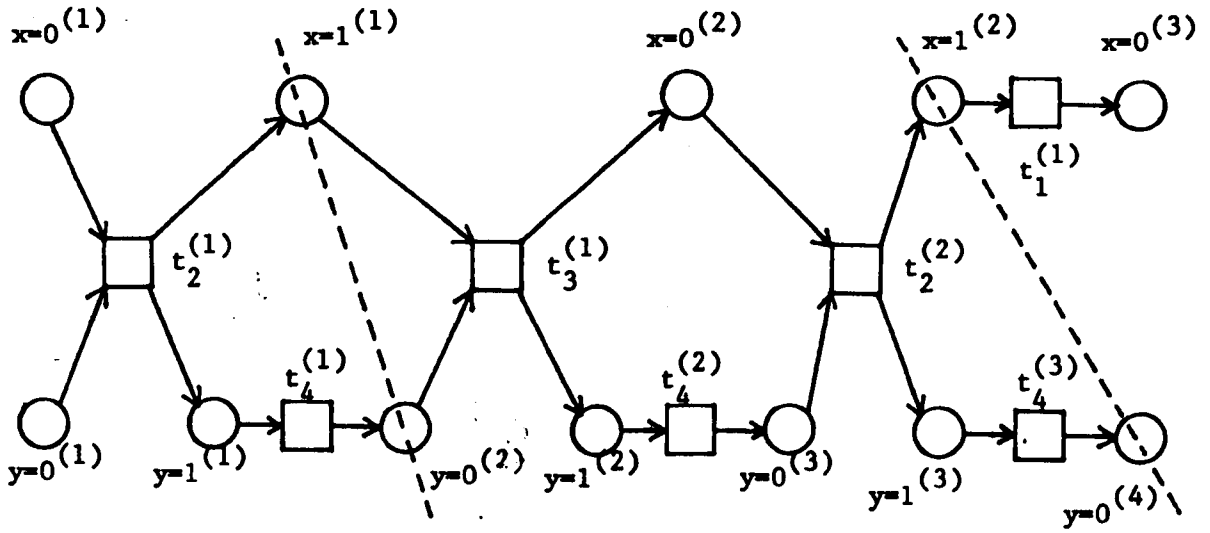


Figure 3.3

We call such a representation of an execution an "occurrence net". To be quite precise, only the "net" part of this, without the inscriptions, is usually called an occurrence net [41]; if the inscriptions, i.e. the links between the occurrence net and the system from which it originates, are taken into account as well, then the resulting structure is called a "process net" [41]. We shall at present, however, find little cause for distinguishing sharply between the two (but see the discussion in section 5.3.3).

The occurrence net depicted in Figure 3.3, then, shows an execution of the original net in which t_2, t_4, t_3, t_4, t_2 occur in serial order, and then t_1 and t_4 concurrently, leading back to the initial state (x,y)=(0,0). In this execution, the choice in state (x,y)=(1,0) (indicated in Figure 3.3 by broken lines) is shown to be resolved both possible ways. First (after the first time (x,y)=(1,0) holds) t_3 is chosen instead of t_1, then (after the second time (x,y)=(1,0) holds) t_1 is chosen instead of t_3.

We see that the possibilities of non-deterministic choice which are

apparent in the system net disappear in the occurrence net because the latter records the actual decisions taken. As a result, in an occurrence net no condition will have more than one input event or more than one output event. We also see that by its very construction an occurrence net will not be cyclic.

It is therefore reasonable to define an occurrence net as a net (B,E,F) (where B is the set of "conditions", E is the set of "events" and $F \subseteq B \times E \cup E \times B$ as before) such that the following hold:

$$\forall b \in B: |bF| \leq 1 \;\&\; |Fb| \leq 1 \qquad (3.4a)$$

$$F^+ \text{ is irreflexive} \qquad (3.4b)$$

where F^+ is the transitive (non-reflexive) closure of F (see Appendix A.1.4). (3.4a) reflects our requirement that no condition has more than one input event or more than one output event. (3.4b) requires that an occurrence net be acyclic.

Two relations, both defined on the set of elements $X = B \cup E$, are of particular interest. For any two elements $x,y \in X$ we write

$$x < y \text{ ("x before y") iff } x\, F^+\, y, \qquad (3.5)$$

i.e. iff there is a directed F-chain from x to y;

$$x \text{ co } y \text{ ("x concurrent to y") iff neither } x < y \text{ nor } y < x$$

That these two relations correspond to what one intuitively associates with them can be checked on our example. Their definition is also justified in [96], to which no further words need being added.

Note two properties of the concurrency relation. Firstly, as defined in (3.5) it is symmetric but not necessarily transitive (as declared at the beginning of section 3.1). Secondly, for occurrence nets the concurrency relation can meaningfully be determined from the net alone, without recourse to a marking. This follows mathematically from the special properties (3.4) of occurrence nets, and it indicates that to each occurrence net its "marking class" can be naturally defined.

Occurrence nets have first been studied extensively by Holt and others in [105] and later by Petri in [95]. More recent research (for example, by Janicki [58,59]) has gone into the direction of extending the property that occurrence nets have a natural marking class to more general nets.

This completes our list of definitions and remarks about nets in general. In the remaining two sections of this chapter we concentrate on occurrence nets in particular. In section 3.3 we define "structured occurrence nets", a generalisation of occurrence nets which is of interest in chapter 4; we also give a few basic results about structured occurrence nets. Section 3.4 contains a discussion of some special properties of infinite occurrence nets.

Summary of section 3.2

We define nets, markings, marking sequences and occurrence nets. For occurrence nets, we define the "before" and "concurrency" relations.

3.3 Structured Occurrence Graphs

3.3.1 Notational Prelude

According to formula (3.4a), a condition in an occurrence net may have no more than one input event and no more than one output event. The only case of real interest is when a condition properly connects two events. For the purposes of both this section and chapter 4 it is convenient to ignore "initial" conditions which have no incoming events and "final" conditions which have no outgoing events. We thus postulate the following stronger version of (3.4a):

$$\forall b \in B: |bF| = |Fb| = 1 \qquad (3.4a')$$

Because of (3.4a'), the set B could now equivalently be described as a binary relation over the set E of events. Or, pictorially speaking, all conditions could equivalently be replaced by single arrows

between the two events they connect. We shall in this section and in chapter 4 prefer to use this latter structure, consisting of a set E of events and a relation over events, which we shall call "occurrence graphs".

The reasons for preferring to use occurrence graphs rather than occurrence nets are purely practical ones. For one thing, some definitions can be formulated more conveniently using occurrence graphs, and for another thing, the pictures get smaller. Thirdly, we shall in our characterisation of atomicity in chapter 4 be concerned with the "before" relation $<$ much more than with individual conditions of B.

An "occurrence graph", then, is here defined as a pair (E,B) where E is a non-empty set of events and $B \subseteq E \times E$ is a relation over E (B is again interpreted as the set of conditions). We allow B to be empty, in order to include a single event as a "degenerated" occurrence graph. We define

$e = \text{head}(b)$ and $e' = \text{tail}(b)$ iff $b = (e,e')$.

For occurrence graphs, we drop the requirement (3.4b). That is, we allow occurrence graphs to contain directed cycles. This is because in chapter 4 we make a connection between such cycles and atomic actions; we do <u>not</u> however use these cycles to describe "repetitive" behaviour.

To sum up: acyclic occurrence graphs and occurrence nets for which (3.4a') holds are completely interchangeable objects, subject to the above translation into one another. Occurrence nets are more general than occurrence graphs in that (3.4a) rather than (3.4a') may hold (this is however insignificant as every occurrence net having initial and final conditions can easily be translated into an occurrence graph containing a special "infinite" event). Occurrence graphs, on the other hand, are more general in that they may contain cycles.

These definitions contradict some previous terminology. In [105] "occurrence graphs" have been defined as acyclic occurrence graphs in the above sense. In [82] "occurrence graphs" have been defined as occurrence nets in the above sense. In [95], occurrence nets in the

above sense have been called "causal nets". In [102] acyclic occurrence graphs in the above sense have been called "event structures". In [86] "occurrence nets" have been defined in a sense substantially different from the above.

Our definition of an occurrence net has followed [42], where the concept of an occurrence graph is however not defined. In view of the discord in established terminology described above, I feel justified in giving the notion of an occurrence graph yet another meaning, in particular since very little confusion is likely to arise from this.

Summary of section 3.3.1

Occurrence graphs are defined as, essentially, occurrence nets which may be cyclic.

3.3.2 Motivation

We show on an example the intentions behind the definitions in this section. Consider the following occurrence graph:

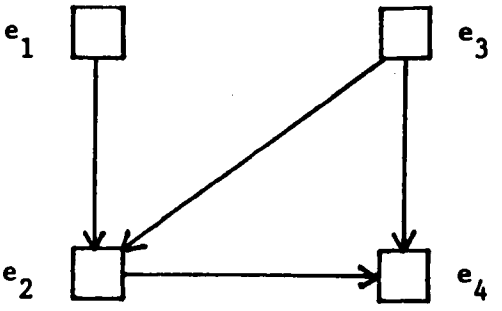


Figure 3.4

We shall consider an atomic action as having the effect of grouping together certain subsets of the events of an occurrence graph and then "collapsing" these subsets into a single event. If atomic actions are

nested (which we allow to be only in a well-nested fashion) then such grouping together and collapsing can also take place in a well-nested fashion.

Let us assume that in Figure 3.4 the events to be collapsed are $\{e_1,e_2\}$ on the one hand, $\{e_3,e_4\}$ on the other hand, and that the result is then to be collapsed altogether. We then obtain the following collapsed versions of our original net, respectively (see Figure 3.5). Graph G_1 arises from collapsing $\{e_1,e_2\}$ but not $\{e_3,e_4\}$; graph G_2 arises from collapsing $\{e_3,e_4\}$ but not $\{e_1,e_2\}$; graph G_3 arises from collapsing both $\{e_1,e_2\}$ and $\{e_3,e_4\}$ simultaneously; finally, graph G_4 arises from collapsing the two events of G_3 into a single event.

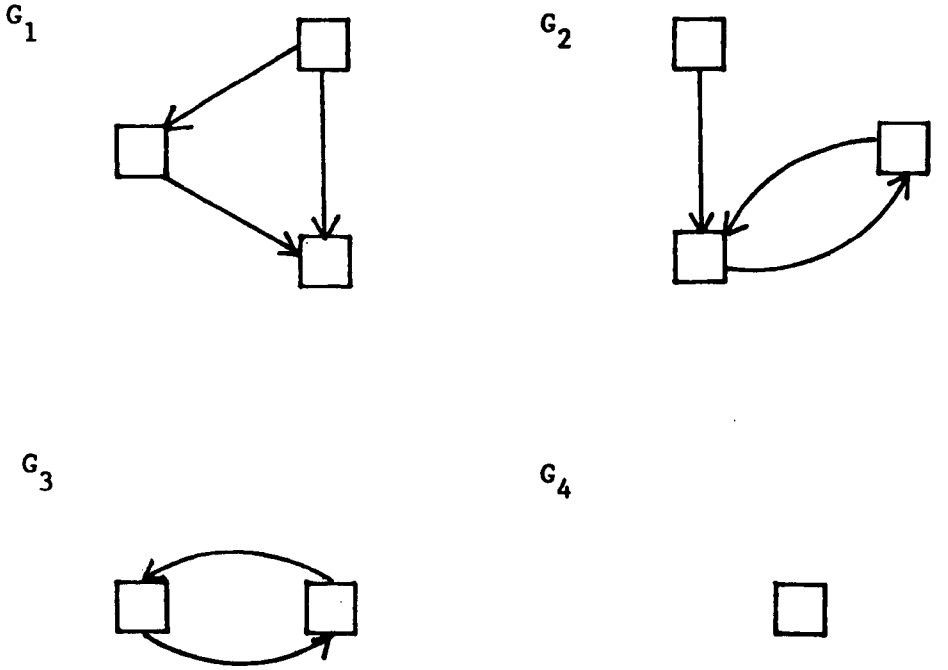


Figure 3.5

We do not interpret the cyclic graphs G_2 and G_3 as indicating the possibility of repetitive behaviour; but rather, we think of all of the

graphs G_1-G_4 as "versions" of the original graph shown in Figure 3.4. All of these graphs describe the same computation, albeit on "different levels of abstraction". The original graph shown in Figure 3.4 is the "basic" graph showing the largest degree of detail. All other graphs are more abstract versions of the basic graph, G_4 being the least detailed, or "most abstract", version.

As we shall take the basic graph as the most detailed description of the computation under consideration, we are not interested in the "interior" of the events in the basic graph. We therefore always assume the basic graph to be acyclic; any cycle in the basic graph would indicate an event being its own cause (which is impossible). This is consistent with later reasoning when cycle-freeness will be taken as a "basicness" criterion (section 4.2).

In subsection 3.3.3 which follows we define the collapsing operation, while in subsection 3.3.4 structured occurrence graphs are defined. The above concepts of different occurrence graphs being associated with different levels of abstraction are also defined in subsection 3.3.4.

Summary of section 3.3.2

Atomicity specifications introduce run-time structure in the form of a structured occurrence graph.

3.3.3 Collapsing of Subgraphs

Let an occurrence graph $G = (E,B)$ and a non-empty subset $E' \subseteq E$ be given. We define the subgraph A generated by E' as the set E' together with all arrows that have both endpoints in E'. Formally,

$$A = (E',B') \text{ where } B' = \{b \in B \mid \text{tail}(b) \in E' \ \& \ \text{head}(b) \in E'\} \quad (3.6)$$

We also denote the set of events E' generating the subgraph A by $\overset{\circ}{A}$. We usually enclose the set E' in a rectangle (see Figure 3.6).

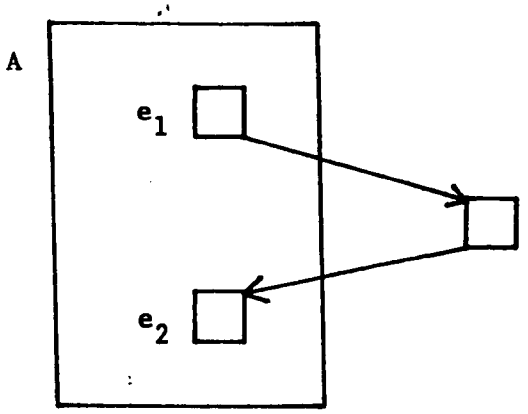


Figure 3.6

As A is again an occurrence graph, all the definitions relating to occurrence graphs can be transferred to subgraphs; in particular, a "before" relationship $<_A = B'^{+}$ can be defined for A which may not coincide with $< = B^{+}$ on A. Note also that A may be disconnected and/or that B' may be empty. For example, in Figure 3.6, A is disconnected, B' is empty and $e_1 < e_2$ in G but not $e_1 <_A e_2$ in A.

We define the "collapsing" of A as the construction of a new graph G[A] from G such that A is replaced by a single new event and all arrows leading into and out of A are replaced by arrows ending and starting, respectively, with the new event. We assume the new event to be uniquely named and call it "A" for the purpose of this definition. Formally, G[A] = (E[A],B[A]) where

$$E[A] = (E \setminus \mathring{A}) \cup \{A\} \tag{3.7}$$

$$\begin{aligned} B[A] = {} & \{(e,e') \in B \mid e \notin \mathring{A} \ \&\ e' \notin \mathring{A}\} \\ & \{(e,A) \mid e \notin \mathring{A} \ \&\ \exists a \in \mathring{A}: (e,a) \in B\} \\ & \{(A,e) \mid e \notin \mathring{A} \ \&\ \exists a \in \mathring{A}: (a,e) \in B\} \end{aligned}$$

In the remainder of this section we present two simple facts about the collapsing operation. The first one indicates that collapsing does

not tear the graph apart, in the sense that paths leading into and out of a subgraph A in G change into paths ending and starting with A, respectively.

Lemma 3.1 $\exists a \in \mathring{A}: e < a \text{ in } G \iff e < A \text{ in } G[A]$
$\exists a \in \mathring{A}: a < e \text{ in } G \iff A < e \text{ in } G[A]$.

Our next lemma shows that the order of collapsing two disjoint subgraphs is immaterial. We call two subgraphs A and A´ of G disjoint iff $\mathring{A} \cap \mathring{A}' = \emptyset$.

Lemma 3.2 If A and A´ are disjoint subgraphs of G then A´´ is a subgraph of G[A], A is a subgraph of G[A´], and G[A][A´] = G[A´][A].

The proofs of lemmata 3.1 and 3.2 follow immediately from the definition of the collapsing operation.

Summary of section 3.3.3

"Collapsing" formally defined as a consequence of the specification of atomic actions.

3.3.4 Structured Occurrence Graphs and Levels of Abstraction

Let an acyclic occurrence graph G = (E,B) be given which we refer to as the "basic graph", E being the set of "basic events". We define a "tree structure" (or "well-nested structure") over G to be a finite set T of sets of events ($T \subseteq 2^E$) such that

$$E \in T \text{ and } \{e\} \in T \text{ for all } e \in E \tag{3.8a}$$

$$\forall E_1, E_2 \in T: E_1 \cap E_2 = \emptyset \vee E_1 \subseteq E_2 \vee E_2 \subseteq E_1 \tag{3.8b}$$

As an example,

$$T = \{\{e_1\},\ldots,\{e_4\},\{e_1,e_2\},\{e_3,e_4\},\{e_1,\ldots,e_4\}\}$$

is the tree structure discussed in subsection 3.3.2.

The sets in T will be used in chapter 4 to model the executions of atomic actions. We therefore call them "atomic activities" or just "activities". Condition (3.8a) is motivated by the wish to consider both the computation as a whole and all basic events as atomic activities. Condition (3.8b) ensures that nested atomic activities do not overlap.

We call a pair (G,T) where G is an acyclic occurrence graph and T is a tree structure over G, a "structured occurrence graph", and we define its structure tree as follows. The nodes of the tree are the activities in T, and a node E' is called a "parent" of another node E'' iff E' is the smallest superset of E'' in T. As a consequence of (3.8a) and (3.8b),

Lemma 3.3 There is a unique smallest superset for all sets in T except E.

The "parent" relationship therefore defines a tree with root E and leaves $\{e\}$, $e \in E$. For $E' \in T$ we define the set of "sub-activities" of E',

$$\overset{\circ}{E}' = \{E'' \in T \mid E' \text{ is parent of } E''\}.$$

Our next aim is to capture the notion of a structured occurrence graph describing a computation at different levels of abstraction. To this end we define levels of abstraction formally and then associate an occurrence graph with each level. Such a graph describes how the events of this level are related to each other, generalising the remarks made in subsection 3.3.2.

For a given structured occurrence graph (G,T), we call a subset $L \subseteq T$ a "level (of abstraction)" iff

$$E = \bigcup \{E' \mid E' \in L\} \tag{3.9a}$$

$$\forall E_1, E_2 \in L: E_1 \cap E_2 = \emptyset \vee E_1 = E_2 \tag{3.9b}$$

(3.9a) requires that all basic events are considered and (3.9b) requires that none of them is considered more than once. Levels can be visualised as "cuts" through the structure tree. For our example discussed in section 3.3.2 we derive the following structure tree and five levels of abstraction $L_0,\ldots,L_4$ (where for simplicity the leaves of the tree are labelled with the names of the basic events they represent):

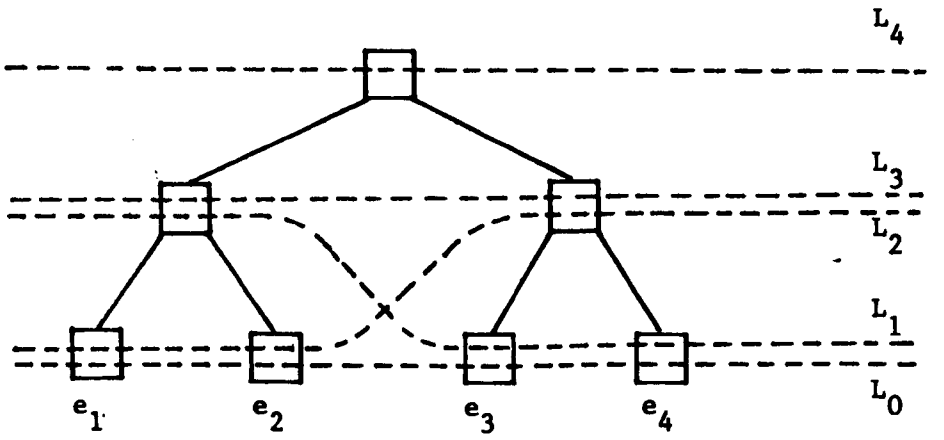


Figure 3.7

We further define:

the "basic level" $L_0 = \{\{e\} \mid e \in E\}$,
the "most abstract level" $L_{top} = \{E\}$, and
for any $E' \in T$ the level $L_{E'} = \{E'\} \cup \{\{e\} \mid e \in E \setminus E'\}$
containing E' and all basic events outside E'.

We define $L' \sqsubset L$ for two levels L' and L iff

$$L' = (L \setminus \{E'\}) \cup \hat{E}' \quad \text{for some } E' \in T,$$

i.e. iff L' arises from L by substituting the sub-activities of E' for E'. We also write $L' = [E']L$ in this case; for instance, $L_1 = [E_2]L_3$ in Figure 3.7. We call L "more abstract" than L' iff $L' \sqsubset L$, where $\sqsubset$ is the transitive closure of $\sqsubset$.

Lemma 3.4 The $\sqsubset$ relationship turns the set of levels into a lattice with L_0 as the minimal element and L_{top} as the maximal

element.

For our example we have the following lattice:

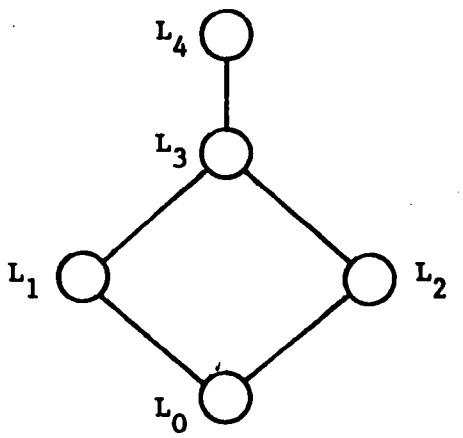


Figure 3.8

Finally we define the occurrence graph associated with a level L by induction over the lattice of levels as follows.

(01) The graph associated with L_0 is the basic graph.

(02) Whenever $L' = [E']L$, G' is the graph associated with L' and A is the subgraph of G' generated by $\overset{\circ}{E}'$, then the graph associated with L is $G'[A]$. We give the new event of $G[A']$ the name "E'", so as to make step (02) repeatedly applicable; the events of the graph associated with L are thus just the activities in L.

As a consequence of lemma 3.2 which shows that the order of collapsing disjoint subgraphs is immaterial and the requirement that all activities be non-overlapping, we have:

Lemma 3.5 (01) and (02) properly define an occurrence graph for each level.

For our example discussed in subsection 3.3.2 we obtain the five level

graphs depicted in Figures 3.4 and 3.5.

Due to the association of an occurrence graph with every level, all concepts defined for occurrence graphs (the $<$ relationship, for example) now become level-dependent. We use the phrase "at level L" in order to avoid confusion about which level is meant.

If the convention of regarding basic events as trivial subgraphs is introduced, a one-to-one relationship between activities $E' \in T$ and subgraphs (generated by $\overset{0}{E}{}'$ if E' is non-basic) can be established. We therefore use the term "activity" for subgraphs $A_1, A_2, \ldots$ as well and extend all definitions accordingly. In particular, $L' = [A]L$ means that A is a subgraph at L' which is collapsed at L; L_A denotes the level containing A and all basic events outside A; and an activity A is said to "contain" another activity A' iff A' is a descendant of A in the structure tree.

Summary of section 3.3.4

We define structured occurrence graphs so as to capture the possibility of hierarchical well-nested collapsing. Derived notions: levels of abstraction, the occurrence graph associated with a level.

3.3.5 Immediate Predecessors and Maximality Axiom

In section 4.4 of chapter 4 we shall wish to be able to identify to an event in an occurrence graph its "immediately preceding" events. When defining the notion of an "immediate predecessor" precisely, care must be taken because our graphs may contain cycles. We devote the present subsection to a suitable definition of "immediate predecessors", showing that for acyclic graphs our definition agrees with the established definition [87].

Usually a node (or an event in our case) e is called an "immediate predecessor" of another event e' if $e < e'$ but for no e'', $e < e'' < e'$. By this definition, in Figure 3.9 below e_1 is not an immediate predecessor of e_3, contrary to our subsequent intentions.

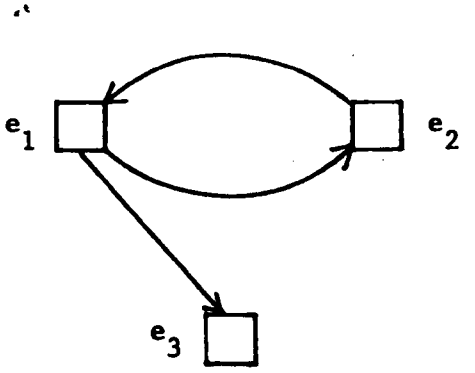


Figure 3.9

We are therefore led to the following different definition of an immediate predecessor.

We call a path from an event e to an event e´ a "(proper) extension" of another path from e to e´ iff the former contains the same events in the same order as the latter, and besides also at least one other event. In Figure 3.9, for example, (e_1,e_2,e_1,e_3) is a proper extension of (e_1,e_3). We call a path "maximal" iff it cannot be properly extended. In Figure 3.9, for example, (e_1,e_2,e_2,e_3) is a maximal path.

We define $e \in E$ to be an "immediate predecessor" of $e' \in E$ (or e´ an "immediate successor" of e), and write $e \lessdot e'$, iff there exists a maximal path $(e_0,\ldots,e_n)$ in which e and e´ are neighbours, i.e. $e=e_i$ and $e'=e_{i+1}$ for some $i \in \{0,\ldots,n-1\}$. Note that by this definition, e_1 is an immediate predecessor of e_3 in Figure 1 because they are neighbours in the maximal path (e_1,e_2,e_1,e_3).

The following lemma shows that this definition agrees, for acyclic graphs, with the usual one referred to above.

Lemma 3.6 (i) $e < e'$ and $\neg\exists e'': e < e'' < e'$ implies $e \lessdot e'$.

(ii) For acyclic graphs, the converse of (i) also holds.

We finally require all occurrence graphs under consideration to satisfy the property that every path can be extended to a maximal path. This is a discreteness property which we subsequently refer to as the "maximality axiom". It is always satisfied for finite graphs.

Summary of section 3.3.5

We define "immediate predecessors" for cyclic nets. The maximality axiom excludes pathological infinite nets.

3.4 K-Density and Bounded Non-Determinacy

In this section we concern ourselves with infinite occurrence nets and with some axioms which serve to exclude certain "unnatural" nets. Our objective is to make a connection between such axioms and the axiom of bounded non-determinacy discussed previously in sections 1.1 and 2.4. This section is self-contained and can be skipped without any loss of continuity.

In particular, we are concerned with Dijkstra's second argument in defence of the axiom of bounded non-determinacy, which states that a program of unbounded non-determinacy would be able to make, within a finite time, an infinite number of decisions. Our reasoning will be that this argument is valid only for sequential programs and that there is no justification in carrying it over to concurrent programs. It follows that this particular argument presents no obstacle against dropping the axiom of bounded non-determinacy for concurrent programs.

As seen in section 3.2, we can define the notion of the "state" of an occurrence net without recourse to a marking. For a given occurrence net (B,E,F), we define a "cut-set" to be a set of elements (of $X = B \cup E$) any two of which are concurrent (see definition (3.5)). A "cut" [42] is defined as a maximal cut-set, i.e. a cut-set which is not a proper subset of any other cut-set. A cut is taken to formalise the notion of a "(global) state". In the net shown in Figure 3.3, for instance, the sets $\{x=0^{(1)}, y=0^{(1)}\}$ and $\{x=0^{(3)}, y=0^{(4)}\}$ as well as the two broken lines

shown, are cuts.

The dual notion to that of a cut is the notion of a "line" [42]. We define a "line-set" as a set of elements such that no two distinct elements are concurrent. A line is then defined as a maximal line-set. Lines are taken to formalise the notion of "sequential subprocesses" of the process under consideration. For example, in Figure 3.3 all "actions on x",

$$\{x{=}0^{(1)}, t_2^{(1)}, x{=}1^{(1)}, t_3^{(1)}, x{=}0^{(2)}, t_2^{(2)}, x{=}1^{(2)}, t_1^{(1)}, x{=}0^{(3)}\}$$

as well as all "actions on y",

$$\{t_2^{(1)}, t_4^{(1)}, t_3^{(1)}, t_4^{(2)}, t_2^{(2)}, t_4^{(3)}, \text{ and corresponding conditions}\}$$

are maximal line-sets, i.e. lines.

For any cut c and line l we have [95]:

Lemma 3.7 $|c \cap l| \leq 1$.

For if there were two different elements x and y in $c \cap l$ then both x co y and (x<y or y<x) which contradicts (3.5).

If the above intersection is always non-empty (i.e. iff $|c \cap l| = 1$ for all cuts c and lines l) then the occurrence net is called "K-dense" [95]. K-density has been proposed in [95] as an axiom to be satisfied by every occurrence net of interest. The motivation for postulating K-density is that at every "time" (= cut c) the "local state" of any given sequential process (= line l) should be determined uniquely (namely as the unique element in $c \cap l$).

K-density is one of the axioms referred to at the beginning of this section, which serve to exclude certain unnatural occurrence nets. K-density excludes, for example, the following net (which is supposed to be infinite as indicated by the ellipses):

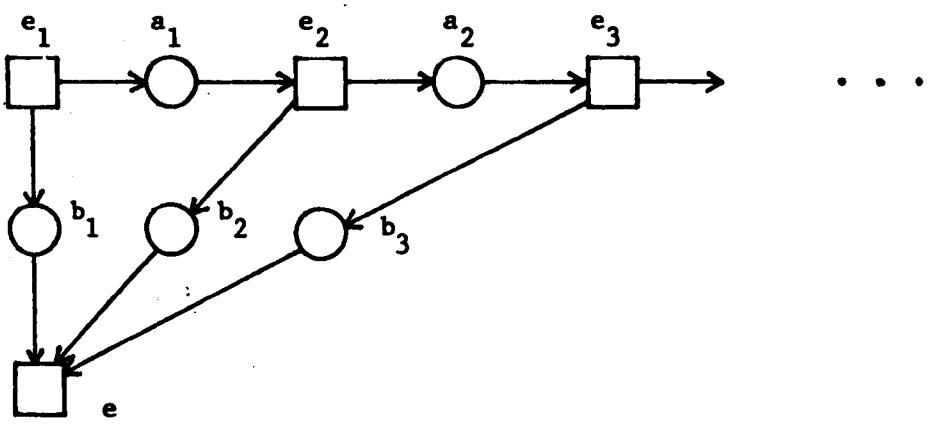


Figure 3.10

The reason why K-density excludes this net is the existence of a cut c_0 and a line l_0 which are disjoint, defined as follows:

$$c_0 = \{b_1, b_2, b_3, \ldots\} \qquad (3.10a)$$

$$l_0 = \{e_1, a_1, e_2, a_2, \ldots, e\} \qquad (3.10b)$$

The net shown in Figure 3.10 and the relationship between K-density and other axioms in general is discussed at length in [13] and [14]. As a matter of interest in the present context, it can be added that K-density and the maximality axiom (section 3.3.5) are not related, in the sense that there are K-dense occurrence nets violating the maximality axiom as well as occurrence nets which satisfy the maximality axiom but violate the axiom of K-density.

There are in fact good reasons (discussed in [103]) not to accept K-density as an axiom. However, whether or not one accepts K-density, it is certainly true that nets such as the one shown in Figure 3.10 should be excluded from consideration (this net not only violates K-density but also the maximality axiom). For, such a net would describe a "sequential process" (namely l_0 of (3.10b)) which is at the same time infinite ($e_1, e_2, \ldots$) and terminates with the distinct event "e" which occurs after all of the e_i.

We postulate that there is no such thing as a "terminating non-terminating sequential process" and that the net shown in Figure 3.10 is not therefore a valid process description. (There is nothing wrong in principle with a "terminating non-terminating" concurrent process: some of its sequential subcomponents may come to a halt while others may not.)

Let us now consider the process envisaged in Dijkstra's second argument, making a choice out of infinitely many possibilities. Implicit in this specification is the requirement that the process always terminates. We envisage a sequential process implementing this specification. For the sake of illustration, assume that there is an infinite succession of choices between "red" and "green" branches as depicted in Figure 3.11:

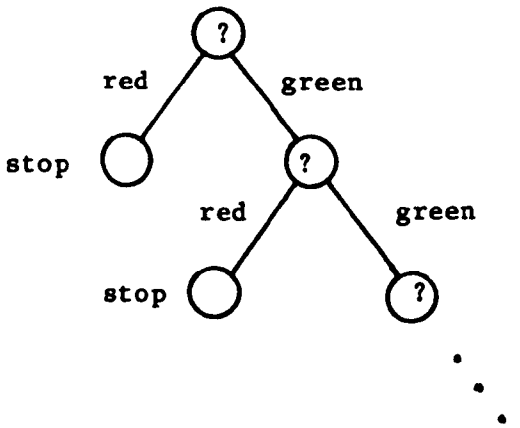


Figure 3.11

When a decision is made in favour of a "red" branch then the process stops; when a decision is made in favour of a "green" branch then the succedding choice is considered next.

The requirement that this process always stops implies in particular that it stops even in case all of its choices are resolved in favour of the "green" branch. But in that case the "stop" event evidently occurs "after" the infinite succession of "green" choices, which exactly

reproduces the occurrence net shown in Figure 3.10. Thus, provided that this latter net is rejected as a "process description", it follows that an infinite number of decisions cannot be made in a finite time by a sequential process.

So far we have reconstructed Dijkstra's argument in more detail. The point of this discussion is to show that the presupposition of the choice process being sequential is essential. We arrived at the above conclusion only by assuming that the choices be arranged in succession; any other arrangement would not necessarily have allowed that conclusion.

Indeed, the unboundedly non-deterministic program

```
(x,y):=(0,0);
<x:=1>  ||  if <x=0> -> <y:=y+1> fi
```

generates, under the fairness assumption, occurrence nets of the form

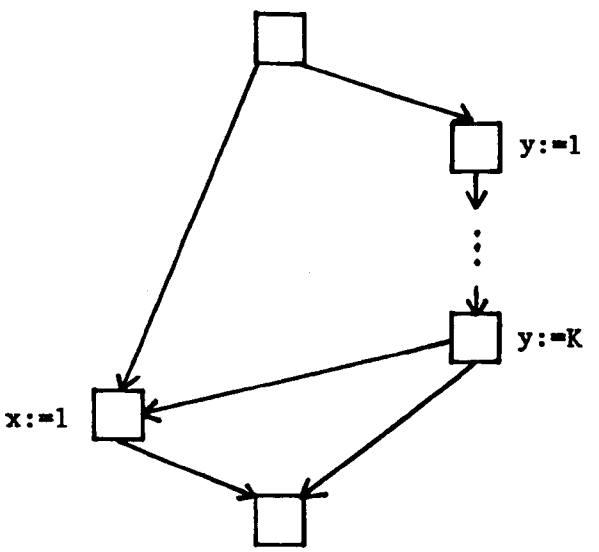


Figure 3.12

where the line-set from "y:=1" to "y:=K" is of variable but finite length. Such an occurrence net is perfectly K-dense and satisfies all other constraints that could reasonably be imposed on occurrence nets [14,103].

This ends our discussion of K-density and its connection to the axiom of bounded non-determinacy. We conclude that Dijkstra's second argument does not seem to stand in the way of considering unboundedly non-deterministic concurrent programs.

Summary of section 3.4

We exclude "terminating non-terminating" sequential processes. The point was made that rejecting such "processes" means accepting the axiom of bounded non-determinacy for sequential processes. However there is no justification for carrying over this argument to concurrent processes.

4. DYNAMIC ATOMICITY CRITERIA

4.1 Introduction and Motivation

In this chapter our aim is to characterise atomic actions in terms of the set of executions they may give rise to. Such executions will be assumed to be describable by a structured occurrence graph as defined in section 3.3 (and as will be spelt out in more detail in section 5.3.3). Under this assumption our characterisation consists of outlawing a certain set of "atomicity-violating" occurrence graphs.

Section 4.2 is devoted to defining the set of "admissible" occurrence graphs. As will be seen, this definition is in essence a generalisation of the "serialisability" criterion [37,106]. This definition is "global" in the sense that an execution as a whole is considered as the basic object of interest. A corresponding "local" atomicity criterion is described in section 4.3. In section 4.4 we discuss the relationship between our atomicity criterion and so-called "two-phase" actions [37,77].

The reason for calling our atomicity criteria "dynamic" is that our basic object of interest is a single execution rather than a program. This can be contrasted with the "static" atomicity criterion discussed in chapter 5 where our objects of interest will be programs rather than executions.

Our dynamic criterion can be thought of as determining the task of an "implementation" of atomic actions. We intend to generalise statements such as the following one, taken from [33]: "Atomic actions ... can be implemented by ensuring between their executions mutual exclusion in time." Our more general version of this statement will be that atomic actions can be implemented by any method ensuring the truth of our atomicity criterion (and we add that actual mutual exclusion is but one of these methods). It is desirable to obtain an overview of such methods because of possible gains in concurrency (especially if the actions are "large").

The present chapter describes the outcome of the pursuit by the author of an idea by Merlin and Randell which can be found in [81] but not in [82]. We recall their idea as a starting point for our motivating discussion. On p. 26 of [81] a "functional activity" is defined as

"... a subnet of an occurrence net in which: (4.1)

1. given any two elements of the subnet, all elements which are on a directed path between these two elements are also members of the subnet;
2. all elements outside the subnet which have outgoing or incoming arcs to the subnet are conditions."

(This quotation has been changed insignificantly. EB).

The reason why functional activities are interesting is stated in [81], p. 27, as follows:

"In fact we regard the concept of a functional activity as generalising and formalising the essence of the notion of an atomic action, ... i.e. an activity which appears `logically instantaneous´ to its environment, and from within which the environment seems `logically unchanging´."

The idea behind (4.1) is to exclude subnets of the kind labelled "A" in Figure 4.1, in which a directed path leads out of A and back into A.

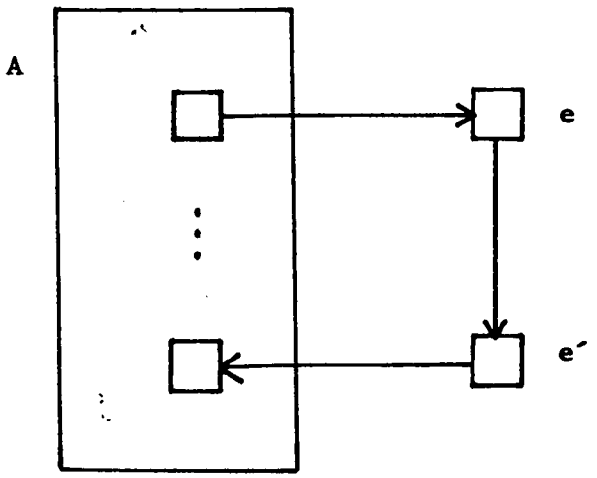


Figure 4.1: A Non-Functional Activity

The reason for excluding A is that the events e and e´ occur strictly after a part of A but also strictly before another part of A. In this way, e and e´ "interfere" with A, and as a consequence, A cannot be considered as an "atomic occurrence" all by itself.

Put differently, it is impossible in the net shown in Figure 4.1 to partition the environment of A into a set of events that can be considered as occurring "before A" and another set of events that can be considered as occurring "after A", with only A "in between" the two sets. This is in contrast to the situation depicted in Figure 4.2 where, even though both e and e´ occur concurrently with parts of A, it is possible to view the original execution as having taken place in the order e,A,e´ (and therefore A as having "occurred atomically"). This is illustrated by the addition of some "insignificant" arrows (shown by broken arrows in Figure 4.2).

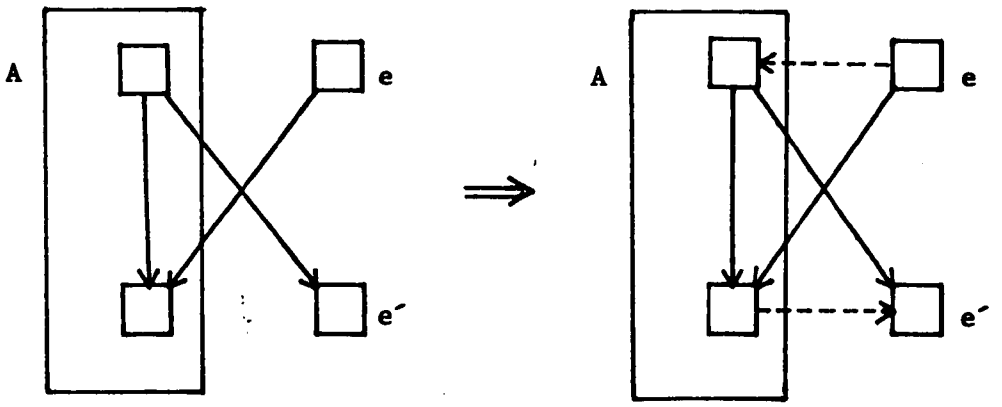


Figure 4.2: A Functional Activity

Note that A is here a functional activity because neither e nor e' lie on a directed path from A back to A.

Put yet another way, we are interested in the property that if A is "collapsed" into a single event (cf. section 3.3.3) then it must be possible to view the resulting net again as a "valid" description of the underlying execution. This is possible in Figure 4.2: if A is collapsed then the strict ordering e,A,e' is obtained. It is however not possible in Figure 4.1: if A is collapsed then a cycle is obtained. We shall take such cycles as indicative of interferences between actions.

Does this discussion correspond to anything which one would intuitively associate with atomic actions? The answer is yes, as we will next show on an example. We need only reconsider the well-trodden problem of the parallel addition [29,88]:

<x:=x+1> || <x:=x+1>

Program P4.1: Parallel Incrementation

Intuitively, as discussed on other simple examples in section 1.1, one would expect P4.1 to be a (precise) implementation of the specification

$$x = x'+2 \qquad (4.2)$$

where (as in section 2.5) the initial value of x is denoted by x'. In section 5.2, we will formally define (4.2) as the relational semantics of P4.1.

Let us now (reasonably) assume that each assignment in P4.1 consists of an event "r" of reading the value of x, followed by an ("internal" and henceforth irrelevant) addition, followed by an event "w" of overwriting the value of x. We call the read/write events corresponding to the left hand assignment "r_1" and "w_1" and the two events corresponding to the right hand assignment "r_2" and "w_2".

The rule of the assignment requires that r_1 always occurs before w_1 and similarly that r_2 always occurs before w_2. Because of the conflict over the common variable x there must be further orderings as well. We introduce the rule that no "write" access may occur concurrently with any other "read" or "write" access to the same variable. In our example this means that only r_1 and r_2 are allowed to occur concurrently while all other pairs of events must occur in some (in general undetermined) order. This rule is a very commonplace one; we discuss it briefly in section 5.4.

We represent the four events r_1, w_1, r_2 and w_2 as events in an occurrence graph. The above rules (i.e. the rule of the assignment and the rule about the exclusion of read/write accesses to common variables) then allow eight possible orderings, four of which are shown in Figure 4.3; the other four may be obtained symmetrically after reversing the order between w_1 and w_2.

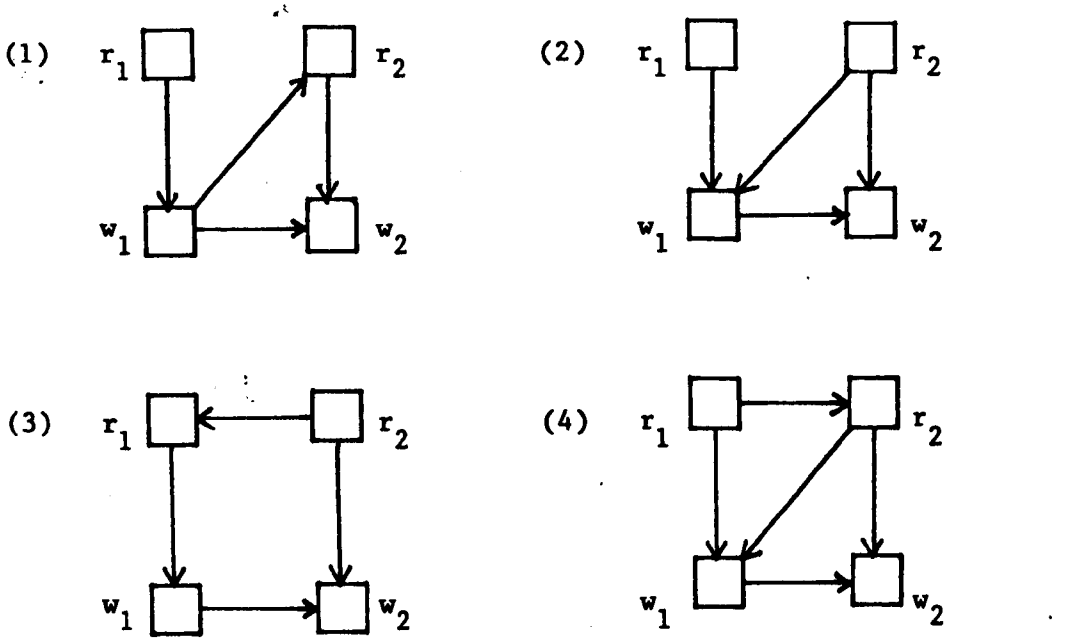


Figure 4.3

Let us consider the four orderings shown in Figure 4.3 in turn. (1) represents a linear ordering r_1, w_1, r_2, w_2. As such it could also correspond to a standard execution of the program

```
x:=x+1; x:=x+1
```

Program P4.2: Sequential Incrementation

which evidently implements (4.2). Ordering (1) therefore represents a valid execution of program P4.1.

Ordering (2), on the other hand, would violate (4.2): the initial value of x would be read concurrently by r_1 and r_2, then w_1 would lead to x being overwritten by this value plus one, whereafter w_2 would lead to x being overwritten again by the same value. Hence the total effect of ordering (2) would be an incrementation of x by 1 rather than by 2 as required. The same is true for orderings (3) and (4) which are effectively only more sequential versions of (2) with the two reads occurring

sequentially rather than concurrently.

Let us consider ordering (2) in more detail, in order to see how the violation of atomicity comes about (see Figure 4.4).

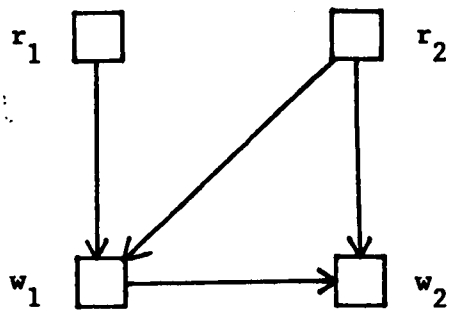


Figure 4.4

The atomicity brackets in P4.1 indicate that the reading and writing of which a single incrementation of x is composed should be considered as two events which "belong together".

Let us first consider the two events r_1 and w_1 in Figure 4.4. Because there is no other "interfering" activity between r_1 and w_1, it is indeed possible to consider them as "belonging together". Therefore it is plausible to consider r_1 and w_1 as an "atomic occurrence".

However, consider r_2 and w_2. In between r_2 and w_2 another activity occurs, namely w_1. As it were, this latter write action invalidates the value read by r_2: either the reading r_2 should have occurred after w_1, or the writing w_2 should have occurred before w_1. As it stands, therefore, r_2 and w_2 together cannot be considered an "atomic occurrence".

We have seen, in sum, that certain occurrence graphs may have to be excluded as a consequence of the atomicity requirements in a program containing atomic actions. We have also seen that in such excluded occurrence graphs it may still be possible that some activity may be viewed as "occurring atomically" while other activity cannot be so

viewed. Our chief objective in this chapter is to make precise this notion of an "atomic occurrence".

As the reader may have noticed, the subgraph A_1 generated by r_1 and w_1 in ordering (2) is a "functional activity" as defined in (4.1), while the subgraph generated by r_2 and w_2 is not a functional activity (see Figure 4.5).

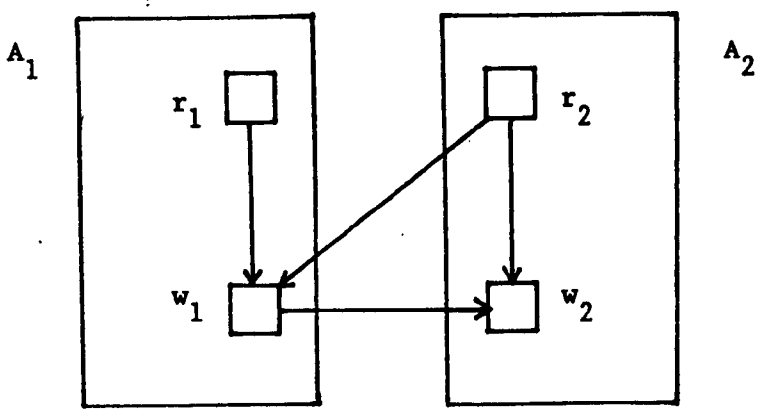


Figure 4.5

This not only shows that functional activities may indeed correspond to what one would intuitively associate with atomic actions, but it also shows that in this example they already capture precisely the desired notion of an "atomic occurrence".

Next in this introduction we show that, despite the last remark, the notion of a functional activity does not yet (not quite, one might say) suffice to capture "atomic occurrences". We show this on another example. Let us consider the following somewhat contrived (but syntactically and semantically well-defined) program

<x:=y> || <y:=x>

Program P4.3

which (intuitively and according to the semantics of section 5.2.2) has

the effect relation

$$x=y=x' \vee x=y=y'. \qquad (4.3)$$

Again, two reads and two writes can be distinguished in P4.3: r_1 (a read on y), w_1 (a write on x), r_2 (a read on x) and w_2 (a write on y). We consider again an execution of P4.3 which obeys the two rules mentioned above (i.e. the rule of the assignment and the read/write rule on common variables) but violates (4.3) (see Figure 4.6).

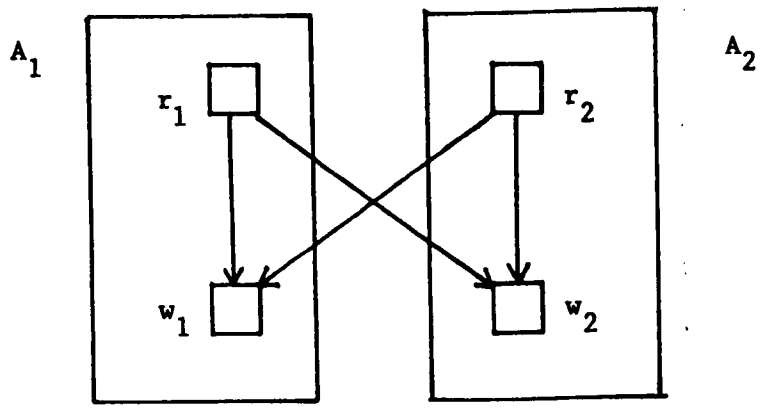


Figure 4.6

The particular execution shown in Figure 4.6 causes the values of x and y to be exchanged and therefore (except in case x=y initially) a violation of (4.3). It follows that such an ordering has to be excluded as a violation of the atomicity requirements present in P4.3.

We observe, however, that both A_1 and A_2 in Figure 4.6 are "functional activities" in the sense of (4.1). (4.1) cannot therefore be an intuitively correct general definition of an "atomic occurrence". We analyse the last example in some more detail in order to arrive at an improved definition.

We aim at a general definition which allows us, given a subgraph, to determine whether or not this subgraph represents an atomic

occurrence. Such a definition must for both A_1 and A_2 in Figure 4.6 give the result that they are non-atomic occurrences. Our way to characterise a non-atomic occurrence A has been by the property that some other activity "interferes" with A, i.e. occurs strictly after a part of A but also strictly before another part of A.

This characterisation can be generalised for the situation shown in Figure 4.6 as follows. Let us first of all consider A_1. While it is true that neither r_2 nor w_2 individually "interfere" with A_1 in the way described, still the "event" A_2 as a whole can be thought of as "interfering" with A_1. To see this, let us suppose that A_2 but not A_1 is "collapsed":

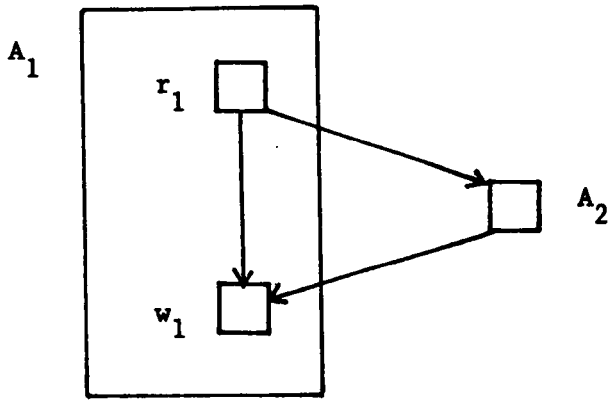


Figure 4.7

Were we ignorant about the internal structure of A_2 then A_2 would appear just as a "single event" which interferes with A_1 in the atomicity-violating way described earlier. Symmetrically, the argument can be made that A_1 as a whole interferes with A_2. It is this property which we shall attempt to capture with our definition of an "atomic occurrence".

This concludes our motivating discussion. Before proceeding, we mention explicitly a few assumptions which we make about our occurrence graph model. We firstly assume that each execution of a concurrent program can be described in detail by an acyclic occurrence graph, called

the "basic graph". We do not assume the basic graph to be unique; we only assume its existence. It is plausible that such a "basic" graph should be acyclic; any cycles would indicate the contradiction that some of the elementary events precede themselves.

Our second assumption is that the specification of an action as atomic can be represented in the occurrence graph as the "collapsing" of the subgraphs it corresponds to. This assumption is evidently reasonable since the collapsing operation allows one to view whole subgraphs as "single atomic events".

Our third assumption is that atomic actions don't "overlap", in the sense that they should lead to a well-nested structure. More concretely, the programmer is not allowed to write "programs" such as the following:

$$<_1\ x:=x-1\ \ ||\ \ <_2\ y:=y-1\ >_1;\ z:=z+1\ >_2,$$

Program P4.4: A Nonsensical Program?

where the angular brackets are supposed to match each other as indicated by their subscripts. This assumption follows the argument contained in [97] where on p. 131 we find the statement that

"... atomic actions, by their very nature, cannot overlap."

We return to P4.4 in section 5.5.

Our fourth (and last) assumption will be that there is always an (implicit or explicit) outermost atomicity bracket around our programs. This assumption does not usually hold true in practice; for instance, none of the programs considered in chapter 6 has such outermost brackets. The assumption is made purely for reasons of convenience and carries no great significance; all subsequent statements can easily be reformulated and remain valid when it is dropped.

Together, these four assumptions allow the executions of all programs under consideration to be described by a structured occurrence graph as defined in section 3.3.4. Henceforth, a structured occurrence graph (G,T) and the "activities" in T (which, according to the last remarks in section 3.3.4, can be viewed as subgraphs) will be our basic objects of interest.

We mention finally two restrictive assumptions which are not made; we also illustrate on an example that it may well be sensible to drop these restrictive assumptions. Firstly, we do not restrict "internal" concurrency within atomic actions. This is in contrast to [37] and [44] where atomic actions ("transactions") are implicitly restricted to be "straight-line" (i.e. loop-free, branch-free and concurrency-free) sequences of more basic actions.

The second assumption we do not make is that of restricted nesting which can often be found in the literature. In line with the reasoning in [97] and [77] we allow an atomic action to contain smaller atomic actions, as well as to be contained in larger atomic actions. Such proper nesting has been prohibited implicitly in [44] and explicitly, but without justification, in [34].

As an example displaying both internal concurrency and proper nesting of atomic actions we consider the program

$$\langle\, \langle x:=x+1\rangle \;||\; \langle x:=x+1\rangle \,\rangle \;||\; \langle x:=2*x\rangle$$

Program P4.5

This program makes sense as a precise implementation of

$$x=2*x'+2 \vee x=2*(x'+2) \qquad (4.4)$$

It may be compared with the following program:

(<x:=x+1> || <x:=x+1>) || <x:=2*x>

Program P4.6

P4.6, being a precise implementation of

$$x=2*x'+2 \vee x=2*x'+3 \vee x=2*(x'+2) \qquad (4.5)$$

indeed differs from P4.5.

I consider the possibility of (well-)nesting atomic actions in an unrestricted way as one of the potentially valuable advantages of atomic actions. In this way, atomic actions can serve to "build new primitives" out of "given primitives". Proper nesting occurs implicitly anyway if one wishes to consider "system-defined" elementary actions as a special class of atomic actions.

Summary of section 4.1

Dynamic atomicity criteria are intended to capture the notion of an "atomic occurrence" as exemplified on two simple examples. We intend to characterise "atomic occurrences" as "occurrences not interfered with by other activity". Executions are assumed to be describable by structured occurrence graphs. No restrictions are imposed on internal concurrency or depth of nesting of atomic actions.

4.2 Global Atomicity Criterion: Serialisability

A straightforward generalisation of our considerations in the previous section leads us to characterise atomicity by prohibiting what has there been called "cycles of interference". For a given structured occurrence graph (G,T) we thus take the characteristic property of atomicity to be that events are partially ordered on all levels of abstraction induced by atomicity specifications (not just the basic level).

Thus we define a structured occurrence graph and the computation it describes to "satisfy atomicity" iff none of its level graphs (see section 3.3.4) contains a directed cycle.

By forming the level graphs for our examples the reader can easily check the intuitive validity of this definition. The structured occurrence graph corresponding to the ordering shown in Figure 4.3(1) does indeed satisfy atomicity, while neither the orderings shown in Figure 4.3(2)-(4) nor the one shown in Figure 4.6 satisfy atomicity. We show the level graphs for Figure 4.6 in full in Figure 4.8 below.

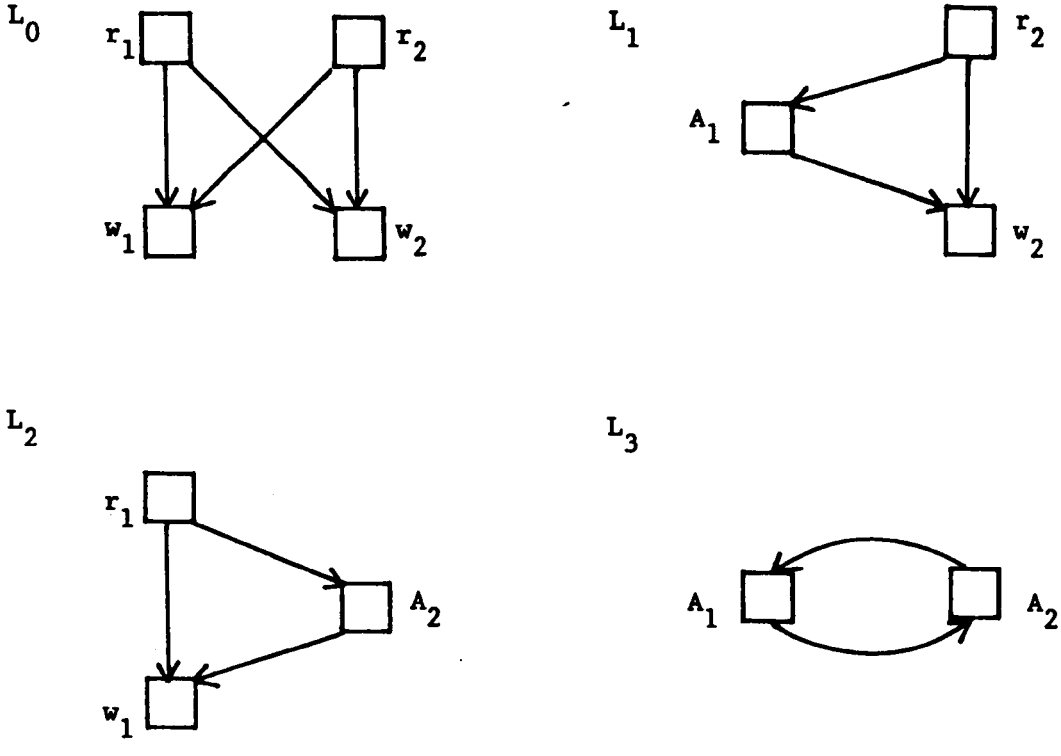


Figure 4.8

The "cycle of interference" perceivable at level L_3 is the reason for this ordering to violate atomicity.

Our definition generalises the so-called "serialisability"

criterion [37,106]. Under some very weak conditions [7] which are here assumed to hold, every partial order can be "serialised", i.e. extended to a linear order. More precisely, for each acyclic occurrence graph $G = (E,B)$ a graph $G_{lin} = (E,B_{lin})$ can be found such that $B^+ \subseteq B^+_{lin}$ and E is linearly ordered under B^+_{lin}. More generally, we have:

Proposition 4.1: A structured occurrence graph (G,T) satisfies atomicity if and only if the basic graph G can be serialised such that the resulting structured occurrence graph (G_{lin},T) describes a linear order on all levels.

Proof: Assuming that (G,T) satisfies atomicity, we may serialise the basic events by processing the structure tree in the following way. Starting with the root of the tree we arrange all subactivities of non-basic activities in linear order, which is possible by assumption. This process stops when all basic events have been reached. Eventually all level graphs describe a linear order.

Conversely, assume that (G,T) does not satisfy atomicity. Then there exists a cycle at some level, the events of which cannot be serialised.

The term "serialisation" is perhaps misleading in that it may suggest that atomicity can only be implemented by actual strict sequencing (i.e. mutual exclusion in time) of the atomic actions of a program. This is not true according to our criterion which allows for the parallel execution of independent atomic actions. Even if atomic actions fail to be independent a partly concurrent execution does not necessarily violate atomicity.

On the other hand, it may be suggested that strict sequencing can always be employed to implement atomicity. However, programs such as the following cannot be serialised, i.e. are not implementable:

```
(x,y) := (0,0);
<x:=1; do y=0 -> skip od>  ||  <y:=1; do x=0 -> skip od>.
```

Program P4.7

A reasonable response would be to equate such programs semantically with "abort", which is what we shall do in section 5.2.2. Typically, in such programs the successful termination of one atomic action depends on the progress of others in a cyclic manner. To prohibit this altogether, it may also seem reasonable to postulate that atomic actions always terminate (this is indeed one of the key axioms in [89]). We shall however find this postulate to be unnecessarily restrictive.

The atomicity criterion defined in this section enables one to tell, given an entire execution, whether or not atomicity is satisfied. This is the reason for our calling it a "global" criterion. The global criterion does not enable us to distinguish between "atomic occurrences" such as A_1 in Figure 4.5 and "non-atomic occurrences" such as A_2 in Figure 4.5. It will be the aim of the next section to describe a "local" criterion which characterises atomic occurrences.

Summary of section 4.2

Serialisability generalised to be our global atomicity criterion. We also generalise the result that every partial order can be extended to a linear order.

4.3 Local Atomicity Criterion: Interference-Freeness

In this section we take a closer look at which activities in an atomicity-violating execution are "interfered with" and which are not. Our aim is to capture precisely the situation that in an atomicity-violating execution (for example, the one shown in Figure 4.9) some of the activities (such as A_1) can be viewed as "occurring atomically"

while others (such as A_2) cannot be so viewed.

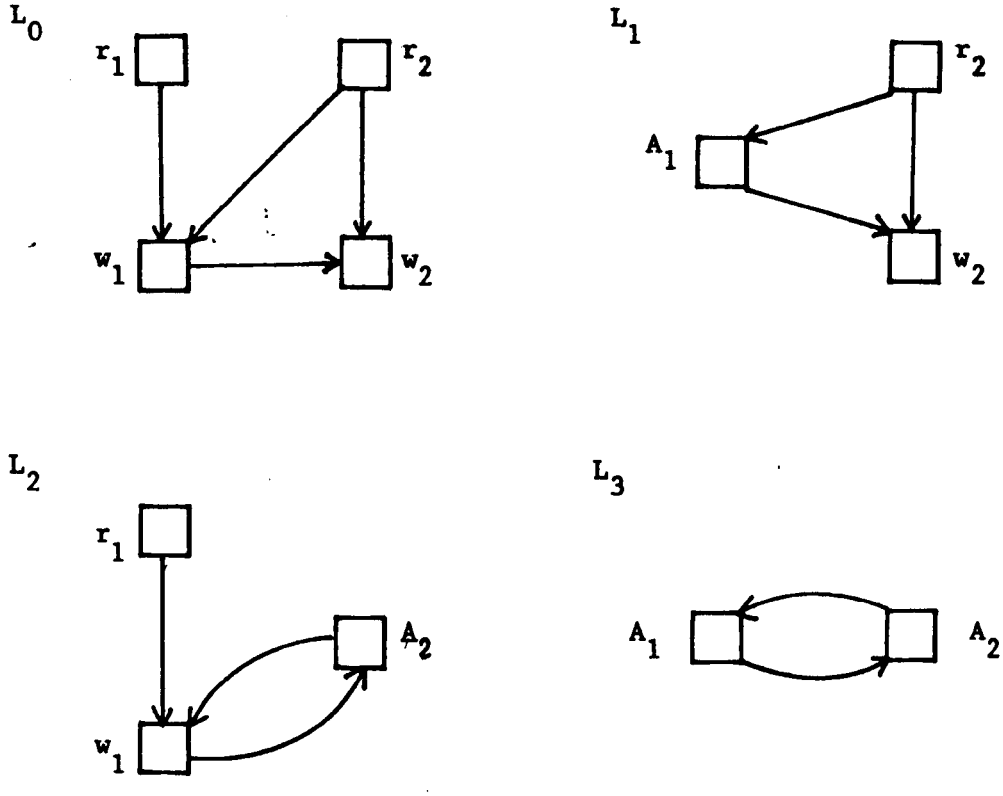


Figure 4.9

We define an event e to "interfere with" an activity A if it occurs strictly after part of A and strictly before another part of A; thus, w_1 interferes with A_2 in Figure 4.9. We shall define A to "occur atomically" if it is not interfered with in this fashion.

Interference of e with A can also be characterised by the fact that at some level L, e and A stand in a cyclic relationship which disappears when A is opened, i.e. at [A]L. For example, w_1 and A_2 are in a cycle at L_2 (cf. Figure 4.9) which disappears when A_2 is opened, i.e. at $L_0 = [A_2]L_2$.

By way of generalisation, we are led to the following general

definition of an atomic occurrence. In a given structured occurrence graph (G,T),

(A1) Basic events occur atomically.

(A2) An activity A occurs atomically iff

(a) $\forall a \in \mathring{A}$: a occurs atomically, and

(b) for all levels L, whenever $e < A < e$ at L then $\exists a \in \mathring{A}$: $e < a < e$ at [A]L.

This definition is inductive in parts (A1) and (A2a), in order to disallow activities to be atomic even though subactivities are not atomic.

Clause (A2b) reflects our above requirement that whenever e and A stand in a cyclic relationship at L then this relationship must not disappear when A is opened, i.e. at [A]L. If (A2b) is violated for some event e and level L then e is one of the "outside activities" that interfere with A, making it non-atomic. The reader can easily check that this is in agreement with our intuitive discussions so far.

Before exhibiting the relationship between the "local" criterion (A1)-(A2) and the global criterion defined in the previous section, I would like to point out a certain subtlety in (A2b). If $e < A < e$ at L then there is at least one cycle at L leading from e back to e. The requirement (A2b) does not exclude one or more of these cycles to be broken when A is opened, but rather it requires that at least one of these cycles remains intact. The definition in [11] is ambiguous on this count. We show an example to underline this point (see Figure 4.10).

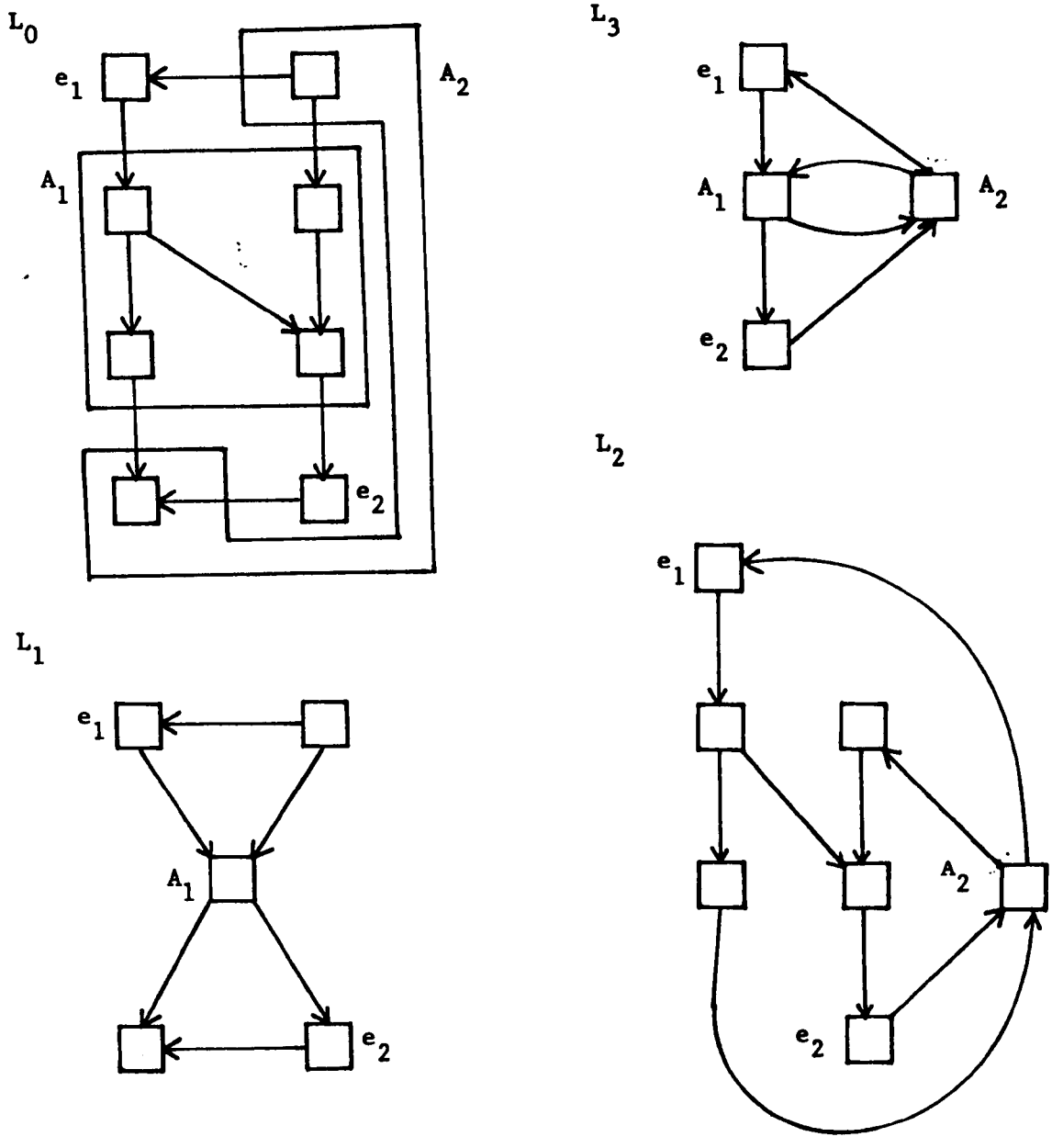


Figure 4.10

In this example it is all too obvious that A_2 is not an "atomic occurrence". However, A_1 is a perfectly atomic occurrence, both intuitively and according to our definition (A1)-(A2). Nevertheless, A_1 is

contained in four different cycles at L_3, one of which (namely the one excluding e_1 and e_2) disappears when A_1 is opened, i.e. at L_2.

We finally show that the local criterion (A1)-(A2) and the global cycle-freenes criterion are interrelated, in the way one would expect. We have:

Proposition 4.2: (i) If for no level either A or one of the activities it contains is involved in a cycle then A occurs atomically.

(ii) Let $e < e$ at L;
then $\exists$ A: $e \leq A \leq e$ at L and A does not occur atomically.

Proposition 4.2(ii) is a weak converse of (i); as the example shown in Figure 4.9 demonstrates, the immediate converse of (i) does not necessarily hold true. We also have the following immediate consequence of proposition 4.2:

Corollary: A structured occurrence graph satisfies atomicity if and only if all of its activities occur atomically.

Proof: (i) If neither A nor any of the activities contained in it is involved in a cycle then (A2b) cannot be violated for A.

(ii) Let $e < e$ at L.
Because of the maximality axiom, there exists a maximal simple cycle $(e=A_0,\ldots,A_n=e)$ at L.
Suppose that all of the A_i occur atomically.
This means that there exist $a_i \in \overset{\circ}{A}_i$ such that $a_0 < \ldots < a_n$ and $a_0 = a_n$ at $[A_0]\ldots[A_{n-1}]L$.
Again we choose a maximal simple cycle $(a_0,\ldots,a_n)$ which must consist of sub-activities of the $\overset{\circ}{A}_i$ only (otherwise $(A_0,\ldots,A_n)$ would not itself be maximal).
This argument is thus repeatable and leads to a contradiction because of the cycle-freeness of the basic graph.
Hence for some i, A_i does not occur atomically, q.e.d.

The local atomicity criterion just described in a sense answers a question posed in section 1.1. Since (it was asked) no piece of program ever occurs without taking up actual time, what then is the special property which enables one to look upon an atomic occurrence such as if it did not take up time? We may measure the actual time consumption of an activity (a subgraph) by any reasonable standard (say just by counting the basic events it contains, or more reasonably by measuring the length of the longest "line" it contains). However large this measure may come out to be, the activity in question can still be considered as not taking up any time iff it is an atomic occurrence according to our criterion.

Summary of section 4.3

We define "atomic occurrences" as our local atomicity criterion. We also prove (proposition 4.2) that (i) Absence of interference implies atomic occurrence, and (ii) Presence of interference implies at least one non-atomic occurrence.

4.4 Inherently Atomic Occurrences and Two-Phase Occurrences

As characterised in the previous sections, the atomic occurrence or otherwise of an activity depends not only on its internal structure but also on its environment at large. This is perfectly plausible because interference between activities pertains not to activities in isolation but to the way in which they are interrelated.

On the other hand, let us suppose that the programmer specifies an action as atomic and that there is no system-provided implementation of atomicity. In this case it would be too optimistic to hope that the environment of the action in question is always so well-behaved as not to interfere with its executions. Rather, the programmer should prevent unwanted interference by explicit programming. Typically, a set of system-provided primitives such as semaphores [29], "test and set" instructions [47] or "locks" [37] can be used for this purpose.

Depending on the availability of more or less knowledge about the surrounding environment, more or less sophisticated "lock protocols" can be employed to safeguard an action against unwanted interference. One of the simplest protocols of this kind, which depends on no knowledge about the environment, is the "two-phase lock protocol" of [37]. This protocol, as do others such as the protocol described in [99], ensure atomic occurrences without regard to the environment.

In this section we explore the salient properties of such occurrences which are atomic regardless of the particular form of their environment. We call such occurrences "inherently atomic", or "contractions" for short. The two-phase lock protocol guarantees inherent atomicity by ensuring that every execution of the action in question consists of a "growing phase" followed by a "shrinking phase", whereby the conceptual moment of occurrence lies between the two phases [77]. We consider our definition of a contraction to generalise this property. This claim will be substantiated below where it will be shown that an activity is a contraction iff it contains an "internal state" which can be thought of as the moment of its occurrence. First we define contractions and exhibit their relationship to atomic occurrences.

Our definition can be motivated as follows. Every (maximal) cycle through A must also pass through an immediate predecessor of A and an immediate successor of A. If A is so structured that from every immediate predecessor of A a path leads through $\mathring{A}$ to every immediate successor of A then the opening of A can never break that cycle.

By way of generalisation, we define that in a structured occurrence graph (G,T),

(C1) Basic events are contractions.

(C2) An activity A is a contraction iff

(a) $\forall a \in \mathring{A}$: a is a contraction, and

(b) whenever $e_1 \lessdot A \lessdot e_2$ at L_A
then $\exists a \in \mathring{A} : e_1 < a < e_2$ at $[A]L_A$.

In (C2b) we consider only the level L_A as defined in section 3.3.4.

However in the proof of proposition 4.3 below it will become apparent that if (C2b) holds for L_A then it holds for all other levels as well.

As an example, we reconsider A_1 in Figure 4.11:

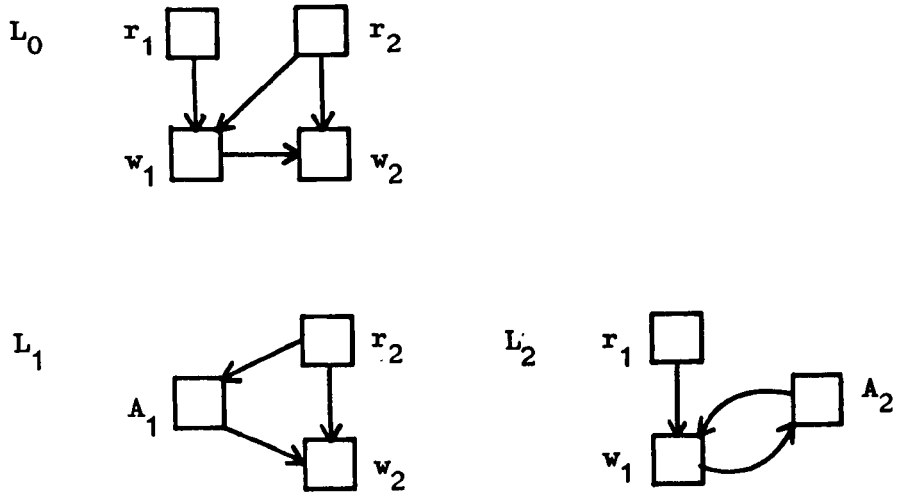


Figure 4.11

At $L_1 = L_{A_1}$ we have $r_2 \lessdot A_1 \lessdot w_2$ and at $L_0 = [A_1]L_1$, $r_2 < w_1 < w_2$. Hence A_1 is a contraction. Its collapsing can be thought of as "contracting" it into w_1 - hence the name. By contrast, $w_1 \lessdot A_2 \lessdot w_1$ at L_2 but $w_1 \not< w_1$ at L_0; hence A_2 is not a contraction.

With the definition (C1)-(C2) we have the following:

Proposition 4.3: (i) Contractions occur atomically.

(ii) If A is not a contraction then a structure T´ can be defined containing the same subtree rooted at A as is contained in T, such that A does not occur atomically in (G,T´).

Proposition 3(ii) is again a weak converse of (i), signifying not that non-contractions occur non-atomically, but that based merely on the internal structure of a non-contraction, nothing can be inferred about its atomic occurrence.

Proof: (i) Let $e \lessdot A < e$ at L.

Because of the maximality axiom, there exists a simple cycle $(e,\ldots,A_1,A,A_2,\ldots,e)$ such that $A_1 \lessdot A$ and $A \lessdot A_2$ at L.

Because $A_1 < A$ at L, by repeated applications of lemma 3.1 one sees that A_1 must contain a basic event d_1 such that $d_1 < A$ at L_A.

Again because of the maximality axiom, there exists a basic event e_1 with $d_1 \leq e_1 \lessdot A$, which is also contained in A_1 (otherwise A_1 would not immediately precede A at L).

Similarly, A_2 contains a basic event e_2 such that $A \lessdot e_2$ at L_A.

Property (C2b) for A requires the existence of an $a \in \mathring{A}$ such that $e_1 < a < e_2$ at $[A]L_A$.

For this a we also have, again by lemma 3.1:

$e \leq A_1 < a < A_2 \leq e$ at $[A]L_A$.

(ii) Since A is not a contraction there exist basic events e_1, e_2 outside A such that for no $a \in \mathring{A}$, $e_1 < a < e_2$ at $[A]L_A$.

We define T' as containing {e} for all $e \in E$, E, the entire subtree rooted at A and the set $\{e_1,e_2\}$.

T' is a tree structure and A does not occur atomically in (G,T').

Proposition 4.3(i) ensures that inherently atomic occurrences are indeed also atomic occurrences (and therefore, by proposition 4.2, a structured occurrence graph containing only contractions is serialisable). Proposition 4.3(ii) signifies that nothing less than the contraction property will do when a programmer wants to ensure atomic occurrences without relying on any system-provided implementation of atomicity. In this way, proposition 4.3 can be seen as a generalisation of the result contained in [37]. As mentioned above, there may be a multiplicity of ways to ensure the contraction property, depending on more or less knowledge about the system; the authors of [99], for instance, are able to exploit the property that the data base model they consider is hierarchical, in order to obtain a lock protocol which is more efficient than the simple two-phase protocol.

We have seen that in the structured occurrence graph shown in Figure 4.11, A_1 is a contraction and the event w_1 can be thought of as the conceptual moment of the occurrence of A_1. As announced above, we now show in general that it is characteristic for a contraction to contain a "state" which can be thought of as the moment of its occurrence. The example shown in Figure 4.12 serves to illustrate this point:

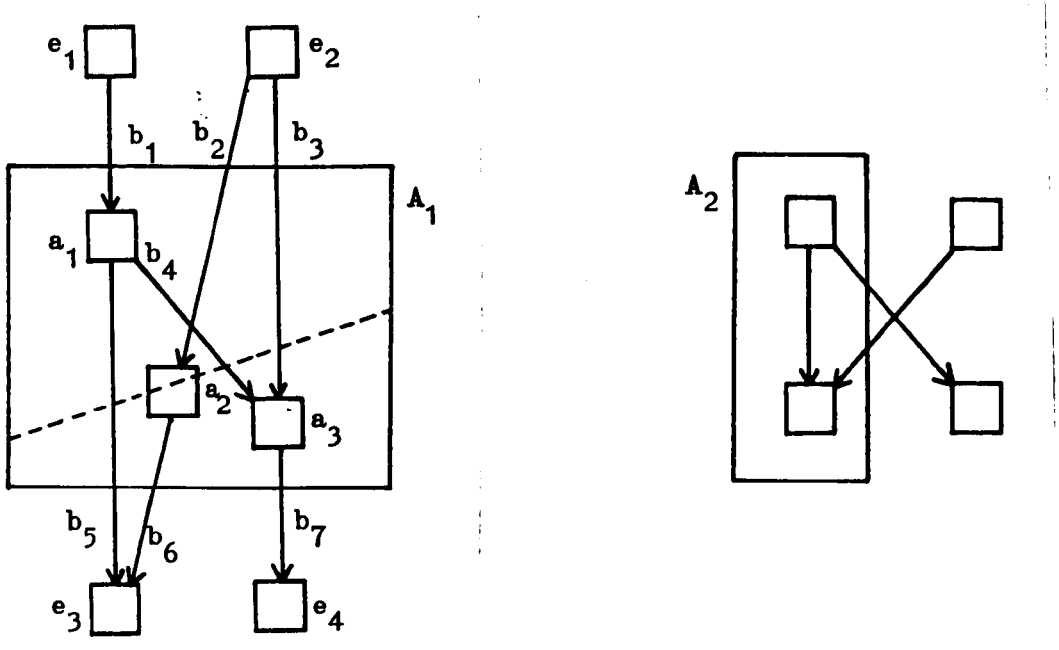


Figure 4.12

In this example, A_1 is a contraction while A_2 is not. The broken line through A_1 represents a "cut" with the property that from every immediate predecessor of A_1 (e_1 or e_2) to every immediate successor of A_1 (e_3 or e_4) there is a path which crosses this cut. No cut with this property can be found for A_2. As in section 3.4 we interpret cuts as "internal states" of an activity and we go on to show that an activity A is a contraction iff it contains a cut with the property just mentioned.

To define internal states formally, let an occurrence graph $G = (E,B)$ and a subgraph $A = (E',B')$ of G given. We first define

$$B^{>} := \{b \in B \mid \text{tail}(b) \notin \mathring{A} \;\&\; \text{head}(b) \in \mathring{A}\}$$

(the set of arrows leading into A), and

$$<_B := \{b \in B \mid tail(b) \in \mathring{A} \ \&\ head(b) \notin \mathring{A}\}$$

(the set of arrows leading out of A).

$B^>$ and $<_B$ can be considered the interface between A and its environment. The relation $<_A = B^{-+}$ (see section 3.3.3) can be extended to elements of the set

$$X = \mathring{A} \cup B^- \cup B^> \cup <_B:$$

if $x,x' \in X$, define $x <_A x'$ iff a directed path inside A leads from x to x'. Two elements $x,x' \in X$ are said to be "A-concurrent" iff neither $x <_A x'$ nor $x' <_A x$. We call a subset $C \subseteq X$ an "A-state" iff its elements are pairwise A-concurrent and it is a maximal set with this property (for instance, the A_1-state shown in Figure 4.12 is $C = \{b_3,b_4,b_5,a_2\}$).

We are now ready to state:

Proposition 4.4: Condition (C2b) in the definition of a contraction can be equivalently replaced by:

(C2b') There exists an A-state C at $[A]L_A$ such that whenever $e_1 \lessdot A \lessdot e_2$ at L_A then $\exists c \in C: e_1 < c < e_2$ at $[A]L_A$.

Proof: (C2b') implies (C2b):

Let an A-state C be given and let $e_1 \lessdot A \lessdot e_2$ at L_A and $e_1 < c < e_2$ at $[A]L_A$ with $c \in C$.

Because C is a subset of X and hence contains only elements in A or bordering on A, one of the following must hold:

either $c \in \mathring{A}$, in which case (C2b) is satisfied with a=c;

or $c \in B$ and $head(c) \in \mathring{A}$, in which case $e_1 < head(c) < e_2$;

or $c \in B$ and $tail(c) \in \mathring{A}$, in which case $e_1 < tail(c) < e_2$.

Conversely, (C2b) implies (C2b'):

Because every path from $b_1 \in B^>$ to $b_2 \in B^>$ must include $tail(b_2) \notin \mathring{A}$, the elements of $B^>$ are pairwise A-concurrent.

We define C_0 as the first A-state including $B^>$; formally,

$C_0 = \{x \in X \mid \forall b \in B^{>} : x$ is A-concurrent to b
and $\neg \exists x' \in X: x' <_A x\}$.

In the example shown in Figure 4.12, $C_0 = B^{>} = \{b_1, b_2, b_3\}$.

The elements of C_0 are pairwise A-concurrent by definition, and C_0 is maximal because no $x'' \in X$ concurrent to all elements of C_0 can have an A-predecessor $x' <_A x''$ in X.

We show that C_0 satisfies the requirements of (C2b').

Let $e_1 \lessdot A \lessdot e_2$ at L_A.

Because A is a contraction,

$a \in \mathring{A} : e_1 < a < e_2$ at $[A]L_A$.

Every path from e_1 to a must contain a pair of neighbours (e_i, e_{i+1}) with $e_i \notin \mathring{A}$ and $e_{i+1} \in \mathring{A}$.

Hence $c = (e_i, e_{i+1}) \in C_0$ and $e_1 < c < e_2$ at $[A]L_A$, q.e.d.

The A-state C which exists by (C2b') can be thought of as a "moment of occurrence" of A. C is by no means unique; in the proof of proposition 4.4, the set C_1 defined as the last A-state including $<_B$ would have done a similar service as C_0. C_0 and C_1 are in fact the "first" and "last" A-states, respectively, which satisfy the property required in (C2b').

Thus, in general, the occurrence of a contraction A can be viewed as consisting of the occurrences of its immediate predecessors, C_0, all intermediate A-states, C_1, and its immediate successors, in that order. In other words, A occurs quasi-sequentially, again illustrating the context-independence of its atomic occurrence.

Summary of section 4.4

We define "inherently atomic occurrences" as a stronger form of atomic occurrences. Inherently atomic occurrences are atomic regardless of the form of their environment. Also, we prove that inherently atomic occurrences are "two-phase" in the sense that their "internal states" could be taken as their "moment of occurrence".

4.5 Discussion

We discuss the extent to which the formal notion of an atomic occurrence defined in section 4.3 can be taken to capture the (an?) intuitive notion of an "atomic occurrence". We split this question into two parts to be discussed separately:

(a) Is every occurrence which satisfies the local atomicity criterion (A1)-(A2) also intuitively an atomic occurrence?

(b) Does every intuitively atomic occurrence satisfy (A1)-(A2)?

Both questions will be discussed in more detail in section 5.3.3 where we will not be forced to rely on an "intuitive" notion of atomicity but will have at hand the relational characterisation which will be defined in sections 5.2 and 5.3. The discussion in 5.3.3 will lead to agreement with the more informal arguments that will be presented in this section.

We consider question (a) first. It seems reasonable to postulate that, indeed, to an intuitively non-atomic occurrence there should be some "outside" activity which interferes with it. This is just what has been prohibited in the definition (A1)-(A2). The motivation given on the occasion of defining (A1)-(A2) therefore quite suffices, I think, to convince the reader that the answer to question (a) should be an unqualified "yes".

Question (b), however, must in general be answered in the negative. In the remainder of this section we show why this is so, but also why one should not be surprised at this being the case. Consider, for example, the program

$$\langle x:=x+1\rangle \;\|\; \langle y:=y+1\rangle$$

Program P4.8: Disjoint Incrementation

whose effect relation is given by

$$x = x'+1 \quad \& \quad y = y'+1 \tag{4.6}$$

Again we assume an execution of P4.8 to consist of two reads (r_1 = a read on x, r_2 = a read on y) and two writes (w_1 = a write on x, w_2 = a write on y).

Consider the following execution of P4.8:

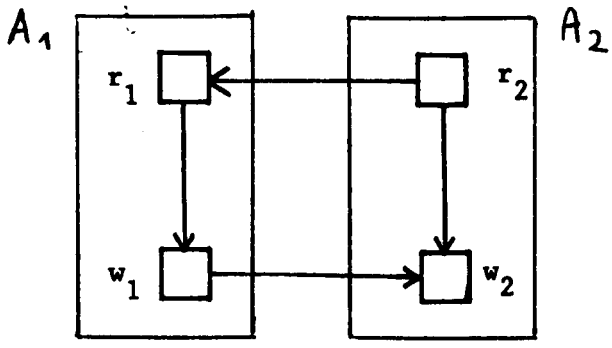


Figure 4.13

Formally, A_2 violates (A1)-(A2) and is not therefore an "atomic occurrence". However, the ordering shown in Figure 4.13 does not violate (4.6) and should therefore be admitted as a valid execution.

The reason for this discrepancy is that the "cross-dependencies" in Figure 4.13 (i.e. the links from r_2 to r_1 and from w_1 to w_2) are not "significant" in the sense that they could be omitted without changing the overall effect of the execution. The property of a dependency being significant cannot be determined from a given occurrence graph alone, which is why it is not surprising that there may exist intuitively atomic occurrences which do not, however, satisfy (A1)-(A2).

The fact that a dependency is insignificant may be obvious, as in Figure 4.13; it may however also be hidden, as we shall show on another example. Consider a program in which variables appear in the right hand side of an assignment but do not contribute semantically to the result, as in:

<x:=x+y+1-y> || <y:=y+x+1-x>

Program P4.9

P4.9 implements, obviously, just the same specification (4.6) as P4.8. However the appearance of x in the second action and of y in the first action will normally (i.e. unless the implementation is "clever") cause cross-dependencies between the two actions which are insignificant.

Suppose, for instance, that an execution of the first command in P4.9 consists of two concurrent reads (r_1^x = a read on x, r_1^y = a read on y) followed by a write w_1 on y, while the second command gives rise to two reads (r_2^x and r_2^y) followed by a write w_2 on y. Then the following "atomicity-violating" execution is in fact perfectly harmless:

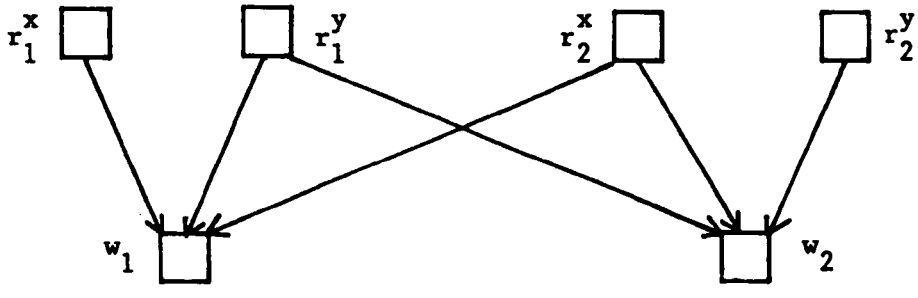


Figure 4.14

Thus, the semantic independence of two actions may not be "visible" in the variables in terms of which the actions have been programmed. A more subtle case of semantic independence is considered in section 5.4. Semantic independence is related to the logic of the program in question, and it is therefore small wonder that the occurrence graph model is not rich enough for it to be captured precisely. Semantic

independence, and for that matter also the property of event dependencies being "significant", is scrutinised further in section 5.4.

5. RELATIONAL SEMANTICS OF PROGRAMS USING ATOMIC ACTIONS

5.1 Introduction

In this chapter our attention will be focussed on the semantic characterisation of atomic actions, and more generally on the semantics of concurrent programs containing atomic actions, in terms of the relational formalism expounded in chapter 2. To begin with, we recall that we conceive of atomic actions as "programming language tools" (section 1.1), on a conceptual par with other linguistic constructs such as, say, the conditional command if...fi. Atomic actions are perhaps most closely analogous to the begin...end brackets of a language supporting block structure. In particular, two programs which agree in everything except the placing of atomicity brackets will in general have to be regarded as two different programs (sometimes, as we shall see in section 6.2, emphatically different).

Why relational semantics? For one thing, even for concurrent programs we are frequently interested in "final states" belonging to certain "initial states". Secondly, relational semantics play a particularly important role in connection with atomic actions. We shall see that if an atomic action is textually replaced by another piece of program with the same overall effect (i.e. the same relational semantics) then the meaning of the program in which the atomic action in question is embedded remains unaffected, no matter how strongly the second program may otherwise differ from the one it replaces. In other words, as far as the environment of an atomic action is concerned, all that matters is the latter's effect relation; or, put still differently, in the calculation of the semantics of a program only the effect relations of its atomic parts need to be taken into account.

We shall develop the relational semantics of atomic actions in two stages. Firstly, we define syntax and semantics of an example concurrent language. This language consists, in essence, of guarded commands augmented by the parallel operator and atomic actions. All of our example programs are written in this language. There will be

restrictions on the way in which atomicity brackets can be inserted within a program; for instance, we continue to disallow the proper overlapping of atomic actions. However, we do allow the proper nesting of atomic actions. Syntax, semantics and correctness criteria for our language will be defined in section 5.2.

Our way of defining the semantics of this language will involve the assumption that the relational semantics of the constituent atomic actions of a given program can be calculated in isolation and then used in the calculation of the semantics of the program itself. Thus, by defining the parallel operator we also very implicitly define the semantic properties of atomic actions.

The second stage in our definition of atomic actions will consist of extracting those semantic properties more explicitly. As described above, we shall be interested in the property that for the environment of an atomic action, only the latter's effect relation carries significance, while its internal details can be abstracted from. We call this the property of "effect-replaceability" and discuss it in section 5.3. We will argue that effect-replaceability is the characteristic "static" semantic property of atomic actions. The replaceability property forms the basis for a, by now, well-known proof method using atomic actions: to prove the invariance of an assertion, one proves its initial truth and then investigates whether every atomic action preserves its truth [33]. This proof method is also discussed in general terms in section 5.3.

Also in section 5.3 we relate our relational characterisation of atomic actions to the "dynamic" characterisation given in chapter 4. We shall in essence show that whenever an execution of a concurrent program "satisfies atomicity" (in the sense of chapter 4) then it relates an initial state to a final state such that the two states stand in the relation prescribed by the relational semantics (in the sense of section 5.2). A weak converse of this result will also be proved. These results depend upon the assumption that every actual execution of a concurrent program can be described by a structured occurrence graph.

The possible presence of "insignificant" event dependencies has in section 4.5 been identified as one of the reasons why the immediate converse of the result outlined in the last paragraph does not hold. The question of which dependencies are significant has in section 4.5 been identified as a question about the semantic interplay between actions and has been postponed until section 5.4. We discuss this problem, to an extent, in section 5.4 where our aim is to find a precise meaning for the notion of the "semantic independence" of actions.

5.2 Syntax, Semantics and Correctness of Concurrent Programs

5.2.1 Syntax

The syntax of a programming language is bound to be influenced by the intended semantics of the programs written in that language. The block structure of Algol 60, for instance, is already reflected in its syntax. Similarly, the syntax of our concurrent programs is guided by an intuitive idea about their meaning. Before giving the syntax in full we first discuss a few design decisions that were made before it was arrived at.

Perhaps at first the most "bewildering" aspect of atomic actions is the extent to which they do not easily fit into the conventional framework of programming language constructs such as conditionals and loops. One can write programs in which atomic actions "overlap" with conditional clauses and loops, and yet have such programs making perfectly good sense. We shall discuss some of the ways in which such overlapping may take place, and we shall then try to understand the principle which causes such at first sight "queer" programs to be well-formed.

Consider as a first example the program

```
1       if <x=1 -> z:=0>
2       [] <x≠1 -> x:=x-1>;
3                  <z:=1>
        fi.
```

Program P5.1

The two atomic actions in lines 1 and 2 of P5.1 mean that the evaluation of the conditions "x=1" or "x≠1", respectively, and the subsequent assignments are to be considered atomic. The atomic action in line 3 is however to be executed only if x≠1 previously, event if "between lines 2 and 3" the value of x has been set to 1 by some other program.

We may wish to allow the separation of the atomic evaluation of a guard and the subsequent command(s). In that case a program such as the following could be written:

```
1       <12x:=x-1;
2       if x=0>1 -> <z:=0>
3       [] x≠0 -> skip>2;
4                 <z:=1>
        fi.
```

Program P5.2

In P5.2 the opening bracket $<_{12}$ has two closing brackets $>_1$ and $>_2$, depending on which alternative becomes elected. It can be appreciated that P5.2, like P5.1, is in principle a well-formed program with a well-defined and unambiguous intuitive meaning.

I would like to point out that in line 2 of P5.2 the atomic action $<\ldots>_1$ terminates with the evaluation of a condition. Intuitively, P5.2 should be so defined as to execute <z:=0> once x has been recognised to be 0, regardless of whether or not the value of x has subsequently been changed by a different program. The evaluation of x to 0 must therefore in some way be understood as committing P5.2 to the first alternative of

the conditional clause, so that there can be no "second thoughts". We shall adjust our semantics (section 5.2.2) to this problem by, in essence, viewing atomic guards as special "commands" with no effects other than to "transfer control".

Matters get still worse when loops are considered. For instance, the following program also makes intuitive sense:

```
1       <1i := 1;
2       do i≠N>12 -> <body>;
3                    <2i := i+1
        od
```

Program P5.3

In P5.3 the second atomic action $<...>_2$ "wraps", as it were, "around the loop", so that in the first iteration the action

<i := 1; i≠N?>

and in all subsequent iterations the action

<i := i+1; i≠N?>

should be regarded as atomic.

One might object that the programmer of P5.3 ought reasonably to make sure that no other component can affect his loop control variable. However we relegate this correctness consideration to be the responsibility of the programmer and do not here let it influence the design of our language. Putting this argument to one side, P5.3 is therefore a perfectly reasonable program.

It is clear that some order must be introduced in this apparent chaos concerning the possible placing of atomicity brackets. For the literature this usually goes without saying. In [69], for instance, only atomic actions of the form

$$\langle B \rightarrow c \rangle \qquad (5.1)$$

are allowed, where B is a guard and c is a command. In [89], on the other hand, we find the constraint that in loops the guard and subsequent commands must be separate atomic actions:

$$\langle B \rangle \rightarrow \langle c \rangle \qquad (5.2)$$

We introduce the syntactic rule to allow, in essence, both (5.1) and (5.2) in our language but none of the somewhat more awkward structures exemplified by P5.2 and P5.3. This rule is dictated by the empiric observation that the forms (5.1) and (5.2) occur frequently in examples, and by the fact that their syntax remains reasonably manageable.

Of course this rule does not lack a certain arbitrariness. However we will now examine the common principle which makes the programs P5.1-P5.3 (and others like them) intuitively well-defined. What is more, we shall tailor the semantics of our language to fit this principle. This ensures that if somebody wanted to generalise our language to include also programs of the type exemplified by P5.2 and P5.3, a few minor changes to the syntax given below and the semantics given in section 5.2.2 could be expected to be sufficient.

We envisage our programs to consist of a number (say m) of sequential components connected by the "parallel" operator. Further, we envisage a complete execution of the entire program to consist, in some sense, of m complete individual executions of its m components. Every execution of a sequential component would in turn simply consist of a succession of the executions of certain atomic actions contained in that component.

Thus, for instance, a general execution of P5.3 could be viewed as a succession of one instance of $\langle \ldots \rangle_1$ followed by zero or more instances of $\langle body \rangle \langle \ldots \rangle_2$. Similarly, an execution of P5.2 could be viewed as either the string $\langle \ldots \rangle_1 \langle z:=0 \rangle$ or the string $\langle \ldots \rangle_2 \langle z:=1 \rangle$, depending on the alternative chosen. The common principle for all of P5.1-P5.3 to make semantic sense we therefore take to be the property

that their executions can be described as strings of atomic actions.

To sum this discussion up, we shall, in general, allow the placing of atomicity brackets in such a way that the above principle is respected; this would permit all of P5.1-P5.3 as valid programs. But we shall, in particular, limit ourselves to programs containing only the structures (5.1) or (5.2) as basic building blocks. Sequential "programs" whose executions cannot be described as strings of atomic actions are outlawed. In particular, the proper overlapping of atomic actions as in P5.4 below is disallowed (see also P4.4 in section 4.1).

$$\langle_1 x:=x-1;\ \langle_2 y:=y-1\rangle_1;\ z:=z+1\rangle_2$$

Program P5.4

The overlapping of atomic actions is further discussed in section 5.5.

We shall add to our formal syntax the rule that every variable (whether "shared" or not) has to be protected by being enclosed within an atomic action. This rule is introduced purely for convenience purposes in defining the semantics of our programs. It will turn out (proposition 5.1 in section 5.2.3) that sequential programs containing atomic actions have the same meaning as sequential programs in which these brackets are omitted. It follows that we can safely omit brackets around "local" variables, and we shall then feel free to do so in subsequent examples.

As in section 2.1, we use capital letters to denote syntactic metavariables. Our metavariables are the following:

PROG: "(concurrent) program"
SEQPROG: "sequential program"
ELPROG: "elementary program"
IF: "conditional clause"
DO: "loop"
GC: "guarded command"
GCLIST: "guarded command list"

V: "variable"

E: "expression"

B: "Boolean expression"

We do not explicitly give the syntax of V, E and B but instead appeal to the reader to add appropriate clauses if he so desires. Our main concern is the semantic definition of the parallel operator rather than the design of a language; we shall not therefore feel completely bound to the syntax which follows but will appeal, whenever applicable, to an obvious extension. For instance, with our syntax a program (such as P4.7 in section 4.2) cannot be written in which a single assignment is followed by a parallel clause. However, it requires but a trivial extension of our syntax and semantics to allow such programs as well.

(SYN1) PROG ::= SEQPROG | SEQPROG||PROG

(SYN2) SEQPROG ::= ELPROG | ELPROG;SEQPROG

(SYN3) ELPROG ::= skip | abort | (a)
V := E | (b)
IF | (c)
DO | (d)
<PROG> (e)

(SYN4) IF ::= if GCLIST fi

(SYN5) DO ::= do GC od

(SYN6) GCLIST ::= GC | GC[]GCLIST

(SYN7) GC ::= <B> → SEQPROG | (a)
<B → SEQPROG> | (b)
<B → SEQPROG>;SEQPROG (c)

Matching non-overlapping pairs of atomicity brackets may be introduced by rules (SYN3e) and (SYN7). We refer to the construct <B> introduced

in (SYN7a) as "atomic condition" or "atomic guard" and to the construct <B→SEQPROG> introduced in (SYN7b,c) as "atomic guarded command". To (SYN1)-(SYN7) we add the further rule that every variable must, at its level of nesting, be enclosed within an atomic action. (The "level of nesting" refers to the possibility of full recursion in (SYN3e).)

Rules (SYN1)-(SYN3a) should be self-explanatory. The appropriate binding precedence will be indicated either textually or by the use of "()" bracket pairs. Rule (SYN3b) deserves the comment that we allow the writing of

<V := E> (5.3)

but not the writing of

<V> := <E> (5.4)

or of assignments in which the expression E is broken up still further into smaller atomic expressions. Of course, the latter programs may also make good sense [67]. However, our previous remarks apply: while we do restrict ourselves to (5.3), there is not the least difficulty in extending our syntax and semantics to the more general case (5.4). The only (obvious) rule to be kept in mind for the assignment is that, contrary to its textual appearance, the evaluation of E has to take place prior to the assignment of the result to V.

Rule (SYN3e) is the place where our understanding of atomic actions comes out in its fullest. We are going to define the relational semantics of a concurrent program in terms of the relational semantics of its constituent atomic parts. In this way we implicitly decree the essential property of an atomic action to be that with respect to its environment only its effect relation is important. Defining the relational semantics of PROG implies (by recursion) that we are safe in assuming to know already the relational semantics of the atomic action <PROG> introduced by rule (SYN3e). We can therefore without problems allow the unrestricted general recursion in this rule. Thus, in the terms of [57], we allow not only "combination" but also "abstraction".

Finally, rules (SYN4)-(SYN7) should again be self-explanatory. In accordance with the language defined in section 2.1 we define a slightly restricted form of loops in (SYN5). The three alternatives in (SYN7) correspond to the various possibilities by which guard constructs of the form (5.1) or (5.2) may, or may not, be followed by further commands within the same guarded command.

As regards variables, we retain the rule that all variables are global integer variables, unless otherwise specified. Occasionally we shall wish to introduce "local" variables whose scope will usually encompass no more than one of the sequential components. Such local variables can easily be translated into global variables with a unique name, whose actual use is limited to their scope (see section 5.2.3).

Summary of section 5.2.1

We define concurrent programs as consisting of sequential components. The sequential components contain atomic actions in such a way that their executions can be described by sequences of atomic actions. We allow full recursion (i.e. nesting) of programs within atomic actions.

5.2.2 Semantics

Let a program c derived from our syntax (SYN1)-(SYN7) be given. Our aim in this section is to define its meaning m(c) as a relation

$$m(c) \subseteq S \times (S \cup \{\underline{|}\}) \tag{5.5}$$

Thus we settle not for the relational semantics (2.1) but for the more general form (2.1′) or, equivalently, (2.28′). This is in the interest of distinguishing "possibly non-terminating programs" from "certainly non-terminating programs" (see section 2.2). As (2.1) can easily be retrieved from (2.1′), we lose nothing; on the contrary, we give for our concurrent programs a semantics which is more general than weakest precondition semantics (see proposition 5.1 in section 5.2.3).

We do not, on the other hand, use the still more general formula (2.28) in order to describe the semantics of c. This is because for the programs considered here we retain the notion that c, when started in an initial state s', either fails to terminate, or else results in precisely one final state (which may not, of course, a priori be determined). The parallel operator || is therefore different from the AND operator discussed in section 2.6 which allows a program to terminate properly in two or more final states. As I have already remarked in section 2.6, I do not at present very well understand the connection between the || and AND operators.

Let us at first consider a parallel program c with two sequential components as our object of interest:

$$c = c_1 \,||\, c_2 \tag{5.6}$$

By analogy with the definition of the concatenation operator, it would be nice if one could deduce the relational semantics of c from the respective relational semantics of c_1 and c_2 only. However, this is not satisfactorily possible. We shall discuss the reasons for this in section 5.2.5 where we relate our way of defining (5.5) to the Owicki-Gries method of proving parallel programs.

In our present definition we will not in general pay attention to the effect relations of c_1 and c_2, but rather we will use the effect relations of the atomic actions contained in c_1 and c_2. To motivate our definition, let us take another look at this "interference" which may prevent the effect relations of c_1 and c_2 from being useful. Let us reconsider the by now familiar example of an unboundedly nondeterministic program:

$$\langle x:=1\rangle \;||\; \underline{\text{do}}\ \langle x=0\rangle \to \langle y:=y+1\rangle\ \underline{\text{od}}$$

Program P5.5

Suppose that (x,y)=(0,0) initially and that x has been recognised to be 0 in the loop, i.e. the atomic guard <x=0> has been executed. As we

have said before, x may subsequently be set to 1 and yet the assignment <y:=y+1> should be executed regardless. In fact, "in between" two atomic actions, another component is free to change the entire state. From the point of view of a single component, therefore, the only thing that remains invariant if it is "between" two atomic actions is just this fact: that "control" resides between these two actions and remains there unless changed by the component in question.

We draw two conclusions. Firstly, the points at which interference is excluded are just the atomic actions of a program. Secondly, "in between" atomic actions, "control" is the only invariant with respect to a given sequential component and as such must enter the semantics. "Control" must be defined independently of the existing state variables. More precisely, suppose one component has just completed one of its atomic actions. We need a means of expressing that, regardless of other possible state changes, the control of this component is and resides after the atomic action just completed (until the execution of the next atomic action of that component is begun).

Sometimes [69,88] so-called "auxiliary variables" have been suggested for the purpose of expressing control. We shall discuss this in section 5.2.5 where we will show that, and why, the Owicki-Gries proof method for concurrent programs actually necessitates the use of auxiliary variables. Instead of auxiliary variables, Lamport introduces special predicates ("at..." and "after...") with a similar effect in [67].

None of this work provides us with a ready-made answer to our problem, namely the definition of m(c). We take a different approach and compare it to known approaches afterwards (section 5.2.5). Our means of expressing control will involve so-called "control sequences" which model "possible control sequences" within a sequential component. This approach reflects just what has been identified in section 5.2.1 as the salient property of the well-formed use of atomic actions: namely that the executions of the sequential programs in question can be described as sequences of atomic actions. Further, it also reflects our principle that atomic actions "behave according to their effect relations". The control sequence formalism which we will develop is very akin to a

corresponding formalism of "firing sequences" for path expressions investigated by M.Shields [98,15]. Similarities and differences will be worked out later.

Let us now attack the definition of m(c) for a general concurrent program c with m sequential components:

$$c = c_1 \,||\, \ldots \,||\, c_m \tag{5.7}$$

Because of our full recursion property (see section 5.2.1) we can assume that in c there are no nested atomic actions (and therefore no nested parallel operators either). Keeping our rule that every variable has to be enclosed within an atomic action in mind, $c_1,\ldots,c_m$ can therefore be assumed to be sequential programs consisting of a set of atomic actions; "skip" and "abort" are assumed atomic by default. We refer to the set of atomic actions contained in c by the names $a_1,\ldots,a_k$. Every a_i ($1 \leq i \leq k$) is contained in one and only one of the components c_j, say in "$cpt(a_i)$".

We assume that for each atomic action a_i we already know its effect relation $m(a_i)$. For assignments <V:=E> we have to extend slightly the semantic definition given in section 2.2. This can be done in the obvious way: $(s',s) \in m(V:=E)$ iff the assignment can be properly executed and s is s' with the value of V replaced by the value of E, and $(s',\underline{|}) \in m(V:=E)$ otherwise (we consider only deterministic assignments).

Because we may have atomic guards (introduced by (SYN7)) we also need to define the effect relation of a Boolean B. We define

$$s'm(B) = \begin{cases} \{\underline{|}\} & \text{if the evaluation of B does not terminate properly} \\ \{s'\} & \text{if the evaluation of B terminates properly and } B(s')=\text{true} \\ \emptyset & \text{otherwise} \end{cases} \tag{5.8}$$

Note that by (5.8) the non-termination of B and the falsehood of B are properly distinguished: in the former case, B acts as though effecting a state transition from s' to $\underline{|}$ and in the latter case, s' has no successor state (i.e. $s'm(B)=\emptyset$). In this case (i.e. when the evaluation of B terminates and yields $B(s')$=false) we call B "disabled"; we shall have to fix the behaviour of our programs in this case.

Finally, we have to define the effect relation of a guarded command "B→c" (where c is a SEQPROG) which, by (SYN7b,c), may also be enclosed within atomicity brackets. This can be done as an obvious extension (consider the arrow → as a sequential concatenation) of the rule (5.8) just given. If $B(s')$=false we call the guarded command in question "disabled"; again, we shall have to fix the semantics of our programs in this case.

The definition of m(c) comes in two parts (formulae (5.16a) and (5.16b) below). First (in (5.16a)), we define the conditions for $(s',s) \in m(c)$ when $s \neq \underline{|}$, i.e. we define the conditions for a proper final state s being reachable from an initial state s'. Secondly (in (5.16b)), we define the conditions for $(s',\underline{|}) \in m(c)$, i.e. for c to fail to terminate properly when started in s'. The question of termination involves problems of deadlock and "fairness".

Our aim is to define $(s',s) \in m(c)$ if s can be reached from s' via a sequence of atomic action executions such that in some sense all m components of c have been "validly" and "completely" executed in the transition from s' to s. Let us illustrate this on our example program P5.5. We represent any serial execution in the form of a string in which states and atomic actions alternate. Consider the following execution of P5.5, starting in the initial state (x,y)=(0,0):

(5.9)

$$(0,0)\langle x=0\rangle(0,0)\langle y:=y+1\rangle(0,1)\langle x=0\rangle(0,1)\underline{\langle x:=1\rangle}(1,1)\langle y:=y+1\rangle(1,2)\langle x\neq 0\rangle(1,2)$$

We have here abbreviated "(x,y)=(i,j)" to "(i,j)". It is intuitively clear that the sequence (5.9) is a valid and complete execution of P5.5

and that the pair of states ((0,0),(1,2)) should therefore be contained in m(P5.5). In order to state this formally, we analyse the general properties of sequences such as (5.9). Let us consider a general sequence

$$u = s_0 a_1 s_1 \ldots a_n s_n \tag{5.10}$$

in which states $s_i \in S \cup \{|\}$ $(0 \leq i \leq n)$ and atomic actions a_i $(1 \leq i \leq n)$ alternate.

We identify three properties which must be satisfied for such sequences to be "valid complete executions". Firstly, every atomic action contained in such a sequence must relate its two neighbouring states in accordance with its effect relation. This reflects our understanding that atomic actions "behave according to their effect relations". Formally, this requirement can be stated as follows:

$$\forall i \in \{1,\ldots,n\}: (s_{i-1}, s_i) \in m(a_i) \tag{5.11}$$

It can be checked that (5.9) satisfies (5.11). Note in particular that this is true for the atomic conditions contained in (5.9). Note also that as a consequence of (5.11), $s_i \neq |$ for $1 \leq i < n$.

The second condition we impose on sequences (5.10) in order that they be valid execution sequences is the following. If we "split" (or "project") such a sequence into component subsequences according to the components in c then each subsequence must be a valid "control sequence" of its respective component. This property, to be made formal below, is reminiscent of a similar property in M.Shields' formalism [98]. We check it in (5.9). Consider first the component sequence <x:=1> which has been underlined in (5.9): this evidently is a valid execution of the first component of P5.5. Consider next the remaining non-underlined subsequence in (5.9):

$$\langle x=0\rangle\langle y:=y+1\rangle\langle x=0\rangle\langle y:=y+1\rangle\langle x\neq 0\rangle \tag{5.12}$$

(5.12) is indeed a possible valid execution of the second component of

P5.5. We refer to strings of atomic actions such as (5.12) as "control sequences"; this distinguishes them from "execution sequences" by which we mean strings in which atomic actions and states alternate. We shall refer to the property that the "projections" of our execution sequences must be valid control sequences of the components as the "validity" property.

It is essential that in forming the projections (e.g. in going from (5.9) to (5.12)) we forget about the intermediate states of the original sequence. This reflects the restriction on "control", i.e. the possibility of other components being free to change the state in between two atomic action executions. Let, in general, a sequence $u = s_0 a_1 \ldots a_n s_n$ of the form (5.10) be given. We define

$$proj(c_j,u) \tag{5.13}$$

as the subsequence of $a_1 \ldots a_n$ obtained by deleting all a_i $(1 \leq i \leq n)$ with $cpt(a_i) \neq c_j$.

The validity property can then be formally expressed as

$$\forall\, j \in \{1,\ldots,m\}\colon proj(c_j,u) \text{ is a control sequence of } c_j \tag{5.14}$$

where the notion of a control sequence is still to be formalised. In general, we then call a sequence u a "valid execution" of c iff it satisfies (5.11) and (5.14).

But (5.11) and (5.14) are not yet strong enough to characterise, say, (5.9) as an execution which leads to the <u>final</u> state (1,2). Any subsequence of (5.9) ending with an intermediate state would likewise satisfy both (5.11) and (5.14). We need (as our third property) to capture the further condition that (5.9) is a "complete" execution of all of its components, in the sense that all component programs have been "textually exhausted". For the first component this is obvious: $\langle x:=1 \rangle$ is clearly a complete execution of it. For the loop we require that the loop termination condition $\langle x \neq 0 \rangle$ be explicitly present.

Thus, we call (5.12) a "complete control sequence" of the second component of P5.5, in contrast, say, to the sequence

$$\langle x=0\rangle\langle y:=y+1\rangle\langle x=0\rangle\langle y:=y+1\rangle \qquad (5.12')$$

The distinction is that (5.12') could be part of an infinite execution of the loop if x=0 always, while in (5.12) the loop termination condition $\langle x\neq 0\rangle$ is part of the sequence, explicitly indicating the termination of the loop.

In general, then, we call any sequence $u = s_0a_1\ldots a_ns_n$ a "complete execution" iff it satisfies (5.11) and, in addition, the following property which is stronger than (5.14):

$$\forall\, j \in \{1,\ldots,m\}\colon \mathrm{proj}(c_j,x) \text{ is a complete control sequence of } c_j \qquad (5.15)$$

where, again, the notion of a complete control sequence is still to be defined.

With these definitions we can now give the first part of the definition of m(c). Let $s',s \in S$. Then

$$(s',s) \in m(c) \text{ iff there exists a complete execution } u = s_0a_1\ldots a_ns_n \text{ with } s'=s_0 \text{ and } s=s_n. \qquad (5.16a)$$

We can now concentrate on an individual component c_j of c and define the "control sequences" and the "complete control sequences" of c_j. Before doing so we have to settle what it should mean if in a conditional clause the guards do not cover the input state. We consider, say,

$$\underline{\text{if}}\ \langle B \to c\rangle\ \underline{\text{fi}} \qquad (5.17)$$

In a sequential context there are two meaningful ways of defining (5.17). One possibility (taken in Algol 60) is to equate (5.17) to "skip" in case B is false. Another possibility (taken in guarded commands) is to equate (5.17) with "abort" if B is false. In the

concurrent context a third possibility presents itself, namely the rule that (5.17) cannot be executed if B is false, i.e. is equivalent to a "wait" command. This would make the if clause rather similar to the usual await clause (see for example [88]).

For reasons to be explained below, we settle for the third possibility. Note that there is a definite distinction between waiting and aborting, even though in a sequential context both could be regarded as equivalent. To see the distinction we compare the following two programs.

```
<x := 1>;
(<x := 0>  ||  if <x=0 -> skip> fi)
```

Program P5.6

```
<x := 1>;
(<x := 0>  ||  if <x=0 -> skip> [] <x≠0 -> abort> fi)
```

Program P5.7

Under the "wait" interpretation, P5.6 terminates properly, in contradistinction to P5.7 which may fail to terminate. Under the "abort" interpretation, however, P5.6 and P5.7 are equivalent.

There are several reasons why we prefer the "wait" interpretation rather than any of the other possibilities. Firstly, nothing is lost in the sense that the other two possibilities can easily be programmed explicitly. This is not true vice versa: under any one of the other two interpretations a "wait" can be programmed only by explicitly programming a "busy wait" loop. Secondly, the "wait" is of practical importance. For instance, the semaphore operation P(s) can now simply be programmed as

```
P(s)  =  if <s greater 0 -> s := s-1> fi.
```

A third reason for preferring the "wait" is that, as it will turn out, its semantics are the easiest to formulate in terms of control sequences. (This came as a pleasant surprise to me.)

Analogously, we specify the semantics of a disabled guarded command, say <B→c>, as a "wait". Thus, as promised, we have fixed the behaviour of our programs for disabled guards. Summing up, a disabled *if* clause will be equivalent to a "wait", while a disabled loop guard will (as usual) enable the successful termination of the loop.

We define "control sequences" (abbreviated c.s.) and "complete control sequences" (abbreviated c.c.s) simultaneously as strings of atomic actions. It is understood throughout that the empty string is always a c.s. (but not a c.c.s.) of any program. Let a sequential component c_j be given. We define the set of its c.s. and c.c.s. by induction over its syntactic structure.

Let $c_j = c';c''$ where c' is an ELPROG and c'' is a SEQPROG.
A c.s. of c_j is either a c.s. of c' or a c.c.s. of c' followed by a c.s. of c''.
A c.c.s. of c_j is a c.c.s. of c' followed by a c.c.s. of c''.

For ELPROGs:
A c.s. (c.c.s.) of "skip" is <skip>
A c.s. (c.c.s.) of "abort" is <abort>
A c.s. (c.c.s.) of <V:=E> is <V:=E>
A c.s. (c.c.s.) of <PROG> is <PROG>

Let c_j = *if* GC_1 [] ... [] GC_k *fi*
A c.s. of c_j is a c.s. of GC_l (for some l, $1 \leq l \leq k$)
A c.c.s. of c_j is a c.c.s. of GC_l (for some l, $1 \leq l \leq k$)

Let c_j = *do* GC *od*
Define the infinite string z as $z = r_1 r_2 r_3 \ldots$,
where every r_l, $l \geq 1$, is a c.c.s. of GC.
A c.s. of c_j is either a (not necessarily proper) prefix of z
or a c.c.s. of c_j

A c.c.s. of c_j is a finite string $r_1 \ldots r_k$ followed by <not B>,
where every r_l ($1 \leq l \leq k$) is a c.c.s. of GC
and "B" is the guard in GC (see below).

For guarded commands GC:
Let gc_1 = <B> → c´ where c´ is a SEQPROG
A c.s. of gc_1 is <B> followed by a c.s. of c´
A c.c.s. of gc_1 is <B> followed by a c.c.s. of c´
Let gc_2 = <B → c´>
A c.s. (c.c.s.) of gc_2 is <B → c´>
Let gc_3 = <B → c´>;c´´
A c.s. of gc_3 is <B → c´> followed by a c.s. of c´´
A c.c.s. of gc_3 is <B → c´> followed by a c.c.s. of c´´.

This completes our definition of (complete) control sequences and thereby also our main definition (5.16a).

We make three remarks relating to these definitions. Our first remark is that, somewhat contrary to their name, control sequences should be taken as "textual" or "syntactic" entities, merely expressing the possible points of control within a sequential program. To illustrate this remark, we note that in the program P5.8 (see below) the sequence

<x:=0><x≠0><skip> (5.18)

is a control sequence according to our definition.

```
<x:=0>;
if <x=0> → skip
[] <x≠0> → skip
fi
```

Program P5.8

It is of course contrary to intuition that the sequence (5.18) should be allowed to be a control sequence if P5.8 is considered as a sequential

component on its own. The reason for admitting (5.18) lies in the possibility that the value of x may be changed by another component in between <x:=0> and <x≠0>, in which case (5.18) could be extended to a valid execution sequence of P5.8. The way in which (5.18) is excluded if such a change does not occur is by the additional "semantic" requirement (5.11): no single state can be inserted between <x:=0> and <x≠0> in (5.18) so as to satisfy (5.11).

Our second remark is that the "wait" interpretation for disabled conditional clauses comes in because control sequences other than ones which actually lead to the execution of programmer-defined guard have not been defined. In other words, an execution sequence containing a disabled guard is simply disallowed. If we wished, say, to change the if according to the "abort" interpretation then we would have to add the rule that

$$\langle(\underline{\text{not}}\ \exists j\colon B_j) \rightarrow \text{abort}\rangle$$

can be an additional c.s. (where the B_j are the existing guards). Thus the "wait" interpretation introduces the least number of rules. This is the reason why I have called it the "easiest" to define in terms of control sequences.

Our third remark concerns the connection between the control sequence formalism expounded here and M. Shields' semantic formalism for path expressions [98]. In our case, because we take into account realistic guarded commands, we have slightly more difficulty in defining control sequences, which are otherwise very similar to Mike's "firing sequences". We do not, on the other hand, admit operations which are shared by way of a "handshake" synchronisation, as are allowed in path expressions as well as in CSP [52]. Such handshake synchronisation can in our programs be simulated by interlocked "wait" statements. We do not in this thesis give general rules for such a simulation but we illustrate the correspondence in section 6.4 on an example.

We now turn to the second stage in our definition of (5.5), which is to determine the conditions in which $(s', \underline{|}) \in m(c)$. We are asking

under which circumstances c should be considered as not properly terminating when started in s'. For this we need an overview of the possibilities of non-termination.

There is first of all the possibility that any one of the a_i produces $\underline{|}$ as its output, be it because a_i aborts or because a_i contains an internal infinite loop or an internal deadlock. In such a case we shall consider the component $cpt(a_i)$ in which a_i is embedded as having aborted, and we shall introduce the rule that this entails the abortion of the program as a whole. We can envisage this condition to be definable in some way as before, i.e. provided

$$u = s_0 a_1 \ldots a_n s_n$$

is a valid execution and $s_n = \underline{|}$ then $(s_0, \underline{|}) \in m(c)$.

The rule that the abortion of a single component implies the abortion of the whole program is a rather strong simplification. More refined approaches are possible and are under investigation. In sections 6.3 and 6.4 we shall find examples of programs which "terminate properly" with some components deadlocking. However in order to capture this more formally, the simple form (5.5) of m(c) would no longer be sufficient. How (5.5) could be generalised to distinguish various possible partial deadlock situations is beyond the scope of this thesis.

A second possibility for c to fail to terminate properly is by a deadlock. This can arise if a conditional clause becomes and remains disabled, as for instance in P5.9 in case $x \neq 0$ initially:

if ⟨x=0 → skip⟩ **fi**

Program P5.9

⟨x:=0⟩ || **if** ⟨x=0 → skip⟩ **fi**

Program P5.10

Thus, $(x=1,\underline{|})$ should be in m(P5.9). But $(x=1,\underline{|})$ should not be in m(P5.10). This is because the first component <x:=0> of P5.10 is sometime going to occur, thus enabling the second component.

Apparently, therefore, in diagnosing a deadlock we have to make sure that a currently disabled command does not at some later stage become enabled again. We reflect this by considering "maximal" executions, by which we mean executions which cannot be extended any further by any other atomic action executions. We call a state a "deadlock state" if it can be reached by a maximal execution but is not at the same time a final state; i.e. the execution in question is maximal but not at the same time complete.

The maximal execution in question may also be infinite. Possible infinite executions are the third case in which we would like to define that $(s',\underline{|})$ is contained in m(c). We first extend the notion of an "execution" to the infinite case. Let

$$u = s_0 a_1 s_1 a_2 s_2 \ldots$$

be an infinite sequence in which states and atomic actions alternate. We call u a "(valid) execution" iff the obvious generalisation of (5.11), i.e.

$$\forall i \geq 1: (s_{i-1}, s_i) \in m(a_i)$$

holds, as well as the obvious generalisation of (5.14).

Our objective is to define $(s',\underline{|}) \in m(c)$ if (not iff) there is an infinite execution. Here we have to deal with the finite delay property. To see the problem, consider the following execution of P5.5 (we reproduce P5.5 below):

$$(0,0)\langle x=0\rangle(0,0)\langle y:=y+1\rangle(0,1)\langle x=0\rangle(0,1)\langle y:=y+1\rangle(0,2)\langle x=0\rangle\ldots \qquad (5.19)$$

$\langle x:=1\rangle \;\|\; \underline{do}\ \langle x=0\rangle \rightarrow \langle y:=y+1\rangle\ \underline{od}$

Program P5.5

(5.19) satisfies our definition of a valid execution of P5.5 because it satisfies the generalised properties (5.11) and (5.14); its projection onto the first component of P5.5, for instance, is the empty sequence, i.e. a valid control sequence of the first component. However, we wish to exclude (5.19) as a valid execution because (intuitively) the finite delay property requires that $\langle x:=1\rangle$ occurs "sometime" during that execution. We incorporate this requirement into the "maximality" property and we call (5.19) not maximal because it contains infinitely often a state in which $\langle x:=1\rangle$ could, but does not, actually occur.

Let us collect these definitions together. Let us first consider a finite execution $u = s_0a_1\ldots a_ns_n$. We call u "maximal" iff there is no a_{n+1} and s_{n+1} such that $s_0a_1\ldots a_ns_na_{n+1}s_{n+1}$ is again a valid execution. Complete executions are thus always maximal; the reverse is however not necessarily true. On the other hand, we call an infinite execution $u = s_0a_1s_1a_2\ldots$ "maximal" iff it does not contain an infinite subsequence $s_{i1},s_{i2},\ldots$ of states all of which enable a certain atomic action, say a_k, but there is no actual occurrence of a_k subsequent to s_{i1}. The phrase "s_i enables a_k" is a shorthand for

$$\exists\, s \in S \cup \{\underline{|}\}\colon s_0\ldots s_ia_ks \text{ is a valid execution sequence.}$$

The latter condition is a slight variant of Karp and Miller's "finite delay property" [61], translated into our language. It is bound to be related to Park's "fair merge" [90], but the exact relationship is a matter for future research. Our finite delay property does not impinge on the usual interpretation of the guarded command non-determinism as "erratic" [30]. To see this, consider the program

```
if <y=1> -> <x:=1> fi  ||  do <x=0> -> <if true -> y:=0
                                       [] true -> y:=1
                                       fi>
                           od
```

Program P5.11

Call the right hand if clause "<if>" for the moment. Our finite delay property continues to allow that in <if> the first alternative is always chosen to the exclusion of the second alternative. In other words,

$$(0,0)\langle x=0\rangle(0,0)\langle if\rangle(0,0)\langle x=0\rangle(0,0)\langle if\rangle(0,0)\ldots \qquad (5.20a)$$

is an infinite execution which is perfectly valid as well as maximal (though not complete). Because of the existence of such an execution, we would like to define $((0,0),|) \in m(P5.11)$.

Suppose, on the other hand, that the choice in <if> is resolved alternatingly. In this case we can consider the sequence

$$(0,0)\langle x=0\rangle(0,0)\langle if\rangle(0,1)\langle x=0\rangle(0,1)\langle if\rangle(0,0)\ldots \qquad (5.20b)$$

which is infinite and valid but neither maximal nor complete. The maximality property is violated because (5.20b) contains infinitely often the state (0,1) which enables the first component of P5.11, but no actual occurrence of that component.

This example also shows that in considering "execution sequences" u it does not suffice to define u just as a string of atomic actions. The atomic actions in (5.20a) and (5.20b) are the same ones in the same order, and yet (5.20a) is maximal while (5.20b) is not.

I do not claim the above "maximality" property to be the last word on the fairness problem. Fairness is at present very much a research subject on its own (see for instance [3,22,90,89]), and I believe some time will have to pass until all aspects of the issue can be considered

as clarified. I have introduced the maximality property with two purposes in mind. Firstly, it seems to be sufficiently strong in order to satisfy the intuition expressed in sections 1.1, 2.4 and 3.4, to the effect that programs such as P5.5 always terminate. (The exact relationship between the maximality property defined here and the properties of maximality and K-density defined in sections 3.3.5 and 3.4, respectively, is not at present clear to me.) Secondly, our definition serves to show that fairness can be introduced as a rather straightforward generalisation of the intuitive notion of an execution being maximal in the sense that it cannot be extended by any further action executions.

Sometimes it has been argued (for example, by Apt and Olderog in [3] and by Broy in [22]) that fairness assumptions are "messy" because they lead to the non-continuity of the wp. While it is true that by assuming fairness one leaves the realm of continuous wp, I would argue (see also section 1.1) that not necessarily need this to be worrying, in particular since by giving up one "nice" property, i.e. continuity, one seems to gain another "nice" property, i.e. maximality.

We then finally are in a position to define

(5.16b)

$(s', \underline{|}) \in m(c)$ iff either there is a finite execution $u = s_0 a_1 \ldots a_n s_n$ with $s' = s_0$ and $s_n = \underline{|}$, or there is a maximal execution starting with s' which is not at the same time complete.

The latter requirement encompasses both deadlocks and infinite looping, or any mixture of both. (5.19) is not thereby admitted as an execution of P5.5 because it is not maximal. This means that P5.5 does not have any infinite executions and does not therefore contain $\underline{|}$ as one of its possible output states; this is in accordance with the intuition expressed earlier (sections 1.1, 2.4 and 3.4).

This concludes our definition of m(c). It should perhaps be added that I do not claim m(c) to be a "full semantics" of a concurrent

program c. One might, for instance, be interested in the properties distinguishing a certain infinitely looping program from another such program (any two such programs will have the same relational semantics). Operating systems are often quoted as examples of programs where relational semantics does not matter so much.

If the reader wishes to obtain a fuller semantics for c he may, for instance, switch to sets of executions rather than relations (5.5). Sets of executions are similarly well-defined mathematical objects. However, relations are much easier to handle mathematically and, moreover, fit the "goal-oriented" approach to program correctness adopted here (see section 5.2.4). Furthermore, in connection with atomic actions relational semantics is just appropriate (see section 5.3.2).

Summary of section 5.2.2

We define the relational semantics of concurrent programs in terms of the relational semantics of the atomic actions contained in them and the way in which these are interconnected. The semantic definition (5.16) consists of a "semantic part" (5.11) which requires atomic actions to behave according to their effect relation and a "control part" (5.14) which requires every sequential component to be validly executed. Fairness is also defined and briefly discussed.

5.2.3 Two Remarks

Our first remark is that the semantics given in section 5.2.2 is consistent with the semantics given in section 2.2. Let c be a sequential program without parallel operators (atomicity brackets can then also be omitted) and let m(c) be its semantics as calculated by the firing sequence formalism given in section 5.2.2. Let moreover $m_0(c)$ be the relation

$$s'm_0(c) = \begin{cases} \emptyset & \text{if } \perp \in s'm(c) \\ s'm(c) & \text{otherwise,} \end{cases}$$

for all $s' \in S$. Then we have

Proposition 5.1 $\dot{m}_0(c)$ coincides with the relational semantics of c given in section 2.2.

The proof of proposition 5.1 is straightforward. For a disabled if clause we have $\perp \in s'm(c)$ because of the existence of a maximal non-complete execution. Due to the above formula we then have $s'm_0(c)=\emptyset$, which means that the disabled if acts as "abort", as required.

Our second remark is that, as can be expected, one can safely omit brackets around atomic actions that access only "local" variables. Let us call a variable "local" if it is used only in one of the sequential components c_j. Let us consider an atomic action a_i which accesses only variables local to its component $cpt(a_i)$. Then a_i can be replaced by any finer-grained copy of itself without affecting the overall semantics. For instance, if $a_i=\langle a_i^1;a_i^2\rangle$ and both a_i^1 and a_i^2 access only local variables then a_i can be replaced by $\langle a_i^1\rangle;\langle a_i^2\rangle$ to give an equivalent overall program. The reason for the latter program being equivalent to the former is that the $\langle a_i^k\rangle$ (k=1,2) commute in all executions with any other action of other components, so that, say, if d is an atomic action and

$$s_0\langle a_i^k\rangle s_1 d s_2 \qquad (k=1,2)$$

is an execution, then so is

$$s_0 d s_1'\langle a_i^k\rangle s_2'$$

and, what is more, $s_2=s_2'$.

We state this as the following proposition:

Proposition 5.2 If a accesses only local variables then $m(c)=m(c[a \leftarrow a'])$,

where a´ is any finer-grained copy of a and c[a ← a´] is c with a (textually) replaced by a´. We do not go into the details of the proof of proposition 5.2. The only use we make of proposition 5.2 is to feel free to omit atomicity brackets around local variables.

The above is, in fact, not the only situation in which two actions commute with each other. Whenever two actions are "independent" then they commute; even mutually dependent actions may commute. This and the property of independence are further discussed in section 5.4.

Proposition 5.2 is important in that parts of a component which are local (i.e. access only local variables) may constitute the real time gains through concurrency. It is therefore to be considered how a possible compiler for our language could best be made aware of them. This may be done by declaring shared variables or else by declaring local variables. I do not in this thesis propose any syntactic means for this. As far as the semantics are concerned there is no trouble as long as one can translate all local variables into global ones with a unique name. In our examples in chapter 6, we frequently use the notation $x[c_j]$ or x[j] when referring to "the variable x local to component c_j". Conceptually, the notions of creating and deleting a local variable would thereby no longer apply; rather, the locality of a variable would be expressed by the fact that only a single component accesses it.

Summary of section 5.2.3

The semantics given in section 5.2.2 is consistent with that given in section 2.2. Atomicity brackets around actions which access only local variables can be omitted.

5.2.4 Correctness

Having secured the definition of the relational semantics (5.5) for a concurrent program there is then no difficulty in defining the correctness of a concurrent program c with respect to a goal G. We do this in complete analogy to the definitions of section 2.5. Let a goal

$$G \subseteq S \times S \qquad (5.21)$$

be given (say by a binary predicate involving primed and unprimed quantities). A program c is called "totally correct" (or just "correct") with respect to G iff

$$\forall s' \in Dom(G): \bot \notin s'm(c) \;\&\; s'm(c) \subseteq s'G \qquad (5.22a)$$

On the other hand, we call c "partially correct" with respect to G iff

$$\forall s' \in Dom(G): S \cap s'm(c) \subseteq s'G. \qquad (5.22b)$$

(5.22a) can immediately be recognised as an analogue of (2.14). We have in section 2.5 however not defined any analogue of (5.22b).

Given the semantics of c, the formulation of its correctness has been very easy. This can be compared with the situation a few years ago when one could find statements such as the following by Keller [64]:

> "It is ... true that there is as yet no widely accepted definition of `correctness' for parallel programs."

Summary of section 5.2.4

Partial and total correctness of a concurrent program can be defined by analogy to corresponding definitions for sequential programs.

5.2.5 Relation to the Owicki-Gries Method

The sorts of programs considered here, and their correctness, have also been considered by Owicki and Gries in [88], Lamport in [67] and Owicki and Lamport in [89]. I will in the sequel attempt to show the connections and differences between their approach and mine, and at the same time compare the relative merits of either approach.

The basic question treated in the aforementioned papers is the following. Let a concurrent program

$$c = c_1 \,||\, c_2$$

be given and let some proof rules for c_1 and c_2 be given. What proof rules can then be derived for c? "Proof rule" refers to the attachment of input/output assertions

$$\{P\}\ c\ \{Q\}$$

to the programs in question. We take $\{P\}\ c\ \{Q\}$ to mean what Owicki and Gries have taken it to mean in [88], p. 320:

> "If P is true before execution of c then Q is true after execution of c. Nothing is said of termination; Q holds provided (their emphasis.EB) c terminates." (Quotation changed insignificantly.EB)

That is, $\{P\}\ c\ \{Q\}$ denotes a "partial correctness" statement.

The answer to this question usually (for instance in [88]) runs as follows. Let

$$\{P_1\}\ c_1\ \{Q_1\} \quad \text{and} \quad \{P_2\}\ c_2\ \{Q_2\}$$

be valid statements about c_1 and c_2, respectively. Suppose further that c_1 (c_2) does not disturb the statement about c_2 (c_1, respectively). Then

$$\{P_1 \,\&\, P_2\}\ \ c_1 \,||\, c_2\ \ \{Q_1 \,\&\, Q_2\} \tag{5.23}$$

is a statement about c.

Of course, the property of interference-freeness circumscribed by the phrase "does not disturb" is of key importance. If c_1 and c_2 are fully independent (e.g. operate on disjoint sets of variables) then this property is always satisfied (see for example [51]). If however c_1 and c_2 interact then it is only satisfied for carefully chosen P_i and Q_i; as a rule, the more c_1 and c_2 interact the weaker the P_i and Q_i have to be.

We shall next present an argument which shows that for the general applicability of the Owicki-Gries method, two assumptions are of crucial importance. Firstly, one has to introduce auxiliary variables and attach intermediate assertions which involve these auxiliary variables "between the atomic actions of c_1 and c_2". Secondly, all assertions must be the strongest possible assertions weak enough to satisfy interference-freeness. (In some cases, the assertions are thereby uniquely determined.) It has been pointed out in [88] that auxiliary variables are necessary, but the requirement that all assertions be as strong as possible occurs only very implicitly. If the reader already knows about these two assumptions then he can skip the remainder of this section, except for the last few paragraphs.

Let us consider an example.

$$c_1 \;=\; (\langle x:=x+1\rangle;\ \langle x:=x+1\rangle) \;\;||\;\; c_2 \;=\; (\langle x:=-x\rangle)$$

Program P5.12

Our semantic formalism easily gives

$$m(P5.12) \;=\; (x=-x'-2) \vee (x=-x') \vee (x=-x'+2) \tag{5.24}$$

as the relational semantics of P5.12 (where x', as before, denotes the initial value of x).

Suppose that we want to prove (5.24) by using the Gries-Owicki method. Thus our aim is to prove

$$\{x=x'\}\ P5.12\ \{(x=-x'-2) \vee (x=-x') \vee (x=-x'+2)\} \tag{5.25}$$

as a statement over P5.12. We shall arrive at the conclusion that such a proof is impossible without introducing auxiliary variables.

Let us proceed as though we were trying to derive a proof of (5.25). We are looking for individual proofs

$\{P_1\}\ c_1\ \{Q_1\}$ and $\{P_2\}\ c_2\ \{Q_2\}$

which are interference-free in the sense of [88] and imply (5.25). More precisely, we are looking for P_i and Q_i such that the following hold:

(i) $\{P_i\}\ c_i\ \{Q_i\}$ (i=1,2) are valid statements;

(ii) the two statements do not interfere with each other; and

(iii) $\{x=x'\}$ implies P_1 & P_2, and Q_1 & Q_2 implies (5.24).

Because both c_1 and c_2 terminate, (i) means that the weakest precondition of c_i with respect to Q_i must be implied by P_i:

$$P_i \Rightarrow wp(c_i,Q_i).$$

This gives for c_1 and c_2, respectively:

(i1) $P_1 \Rightarrow Q_1[x \leftarrow x+2]$
(i2) $P_2 \Rightarrow Q_2[x \leftarrow -x]$

where $Q[x \leftarrow E]$ again denotes Q in which all free occurrences of x have been replaced by E.

The obligation (ii) implies that Q_1 must be invariant under the change of sign of x by c_2:

(iii1) $Q_1[x \leftarrow -x] \Rightarrow Q_1$

and that Q_2 is left invariant when x is increased by 1. By induction it follows that Q_2 must be invariant over any increase of x. Formally, if we define

$$Q_{2,k} = Q_2[x \leftarrow x+k] \qquad \text{for} \quad k = 0,1,2,\ldots$$

then we require the truth of

(ii2) $\forall\, k \geq 1 : (Q_{2,k} \Rightarrow Q_{2,k-1})$.

The obligation (iii) can be expressed directly as:

(iii1) $\{x=x'\} \Rightarrow P_1 \,\&\, P_2$
(iii2) $Q_1 \,\&\, Q_2 \Rightarrow (x=-x'-2) \vee (x=-x') \vee (x=-x'+2)$.

Our problem can now be stated as that of finding P_i and Q_i such that (i1)-(iii2) hold true.

We can eliminate P_1 and P_2 by deleting (iii1) and reducing (i1) and (i2), respectively, to

(i1') $\{x=x'\} \Rightarrow Q_1[x \leftarrow x+2]$
(i2') $\{x=x'\} \Rightarrow Q_2[x \leftarrow -x]$

It remains to determine some Q_i satisfying (i1'), (i2'), (iii1), (ii2) and (iii2). Because of (iii2) we want to define the Q_i as strong as possible. We try to do so by successively approximating them by two series $Q_i^{(j)}$, $j=0,1,\ldots$, keeping the $Q_i^{(j)}$ as strong as possible. Our initial choice is

$$Q_1^{(0)} = Q_2^{(0)} = \text{false}$$

which satisfies everything except (i1') and (i2'). To satisfy (i1') and (i2') we weaken the $Q_i^{(0)}$ to

$$Q_1^{(1)} = \{x=x'+2\},\ Q_2^{(1)} = \{x=-x'\}$$

However, now $Q_1^{(1)}$ fails to satisfy (iii1) and $Q_2^{(1)}$ fails to satisfy (ii2). In order to remedy this we have to add to $Q_1^{(1)}$ a term with x replaced by -x:

$$Q_1^{(2)} = \{|x|=x'+2\},$$

and to $Q_2^{(1)}$ we have to add an infinite series of terms with x replaced by x-1, x-2, etc. We thus arrive at

$$Q_2^{(2)} = (\exists k \geq 0 : x = -x' + k).$$

$Q_1^{(2)}$ and $Q_2^{(2)}$ are the strongest choices implied by $Q_1^{(1)}$ and $Q_2^{(1)}$, respectively, which satisfy (iil) and (ii2). Yet they are already too weak to allow the desired conclusion (iii2) because by setting k:=2*x'+2 we see that

$$Q_1^{(2)} \ \& \ Q_2^{(2)} = \{x = x' + 2\}$$

and {x=x'+2} is not a term on the right hand side of (iii2).

We have thus arrived at the conclusion that there can be no direct Owicki-Gries style proof for the statement (5.25) about the concurrent program P5.12. The snag has been our requirement that Q_2 must be invariant over any increase of x by 1. We really only need the weaker requirement that Q_2 be invariant over at most two increases of x. In order to express this, we need an auxiliary variable which counts the number of times x has been increased and thereby expresses the points of control of c_1.

Such auxiliary variables can be introduced in a systematic manner. To see this on our example, I reproduce (by courtesy of E.W. Dijkstra) an Owicki-Gries proof of (5.25). We define two auxiliary program counters i and j (for c_1 and c_2, respectively), both initialised to 0, and amend P5.12 as follows.

c_1: $P_1 = \{i=0 \;\&\; (j=0 \;\&\; x=x' \;\vee\; j=1 \;\&\; x=-x')\}$
$\langle x:=x+1;\; i:=1\rangle;$
$\{i=1 \;\&\; (j=0 \;\&\; x=x'+1 \;\vee\; j=1 \;\&\; (x=-x'+1 \;\vee\; x=-x'-1))\}$
$\langle x:=x+1;\; i:=2\rangle$
$Q_1 = \{i=2 \;\&\; (j=0 \;\&\; x=x'+2 \;\vee\; j=1 \;\&\; (x=-x'+2 \vee x=-x' \vee x=-x'-2))\}$

c_2: $P_2 = \{j=0 \;\&\; (i=0 \;\&\; x=x' \;\vee\; i=1 \;\&\; x=x'+1 \;\vee\; i=2 \;\&\; x=x'+2)\}$
$\langle x:=-x;\; j:=1\rangle$
$Q_2 = \{j=1 \;\&\; (i=0 \;\&\; x=-x' \;\vee\; i=1 \;\&\; (x=-x'-1 \;\vee\; x=-x'+1)$
$i=2 \;\&\; (x=-x'-2 \;\vee\; x=-x' \;\vee\; x=-x'+2))\}$

Owicki-Gries Proof of (5.25)

The assertions of this proof have been constructed as the strongest possible assertions (including program counters) which are weak enough to satisfy the interference-freeness property.

We show that these assertions have to be as strong as possible. To this end, let us reconsider P5.12 with the "interfering" case fixed as a default. We can easily do so by using a variable

var y: 0..1

initialised to 0, as a sequencing device:

c_1' = $\langle (x,y):=(x+1,1)\rangle$; if $\langle y=0 \to x:=x+1\rangle$ fi ||
c_2' = if $\langle y=1 \to (x,y):=(-x,0)\rangle$ fi

Program P5.13

(The ifs in P5.13 are, in accordance with the semantics defined in section 5.2.2, to be interpreted as awaits.) Intuitively, P5.13 transforms $x=x'$ into $x=-x'$, which also agrees with our semantics because of the existence of the following (but no other) execution:

$(x',0)\langle x,y:=x+1,1\rangle(x'+1,1)\langle y=1\to x,y:=-x,0\rangle(-x'-1,0)\langle y=0\to x:=x+1\rangle(-x',0)$

However, let us consider the following statements:

$$\{x=x' \ \& \ y=0\} \quad c_1' \quad \{|x|=x'+2\} \tag{5.26a}$$

$$\{x=x' \ \& \ y=0\} \quad c_2' \quad (\text{true}) \tag{5.26b}$$

Surely, both are valid (partial correctness!) statements because under the assumption y=0 initially neither c_1' nor c_2' by themselves terminate. If the two statements (5.26a) and (5.26b) satisfy the interference-freeness property defined in [88] then the parallel operator rule (5.23) allows us to conclude that

$$\{x=x' \ \& \ y=0\} \quad \text{P5.13} \quad \{|x|=x'+2\}$$

is a statement over P5.13, a result which is manifestly counterintuitive (whether or not taken as a partial correctness statement) and contrary to our semantics.

This result depends, of course, on whether or not (5.26a) and (5.26b) are indeed interference-free. Intuitively they should not be. To enable the reader to judge for himself, I reproduce the Owicki-Gries interference-freeness formulae by quoting from [88], pp. 322-323, temporarily using their paragraph numbers:

"(3.4) Given a proof $\{P\}$ S $\{Q\}$ and a statement T with precondition pre(T), we say that T `does not interfere with' $\{P\}$ S $\{Q\}$ if the following two conditions hold:

(a) $\{Q \ \& \ \text{pre}(T)\}$ T $\{Q\}$

(b) Let S' be any statement within S but not within an <u>await</u>. Then $\{\text{pre}(S') \ \& \ \text{pre}(T)\}$ T $\{\text{pre}(S')\}$.

(3.5) $\{P_1\}$ S_1 $\{Q_1\}$, ..., $\{P_n\}$ S_n $\{Q_n\}$ are `interference free' if the following holds. Let T be an <u>await</u> or assignment statement (which does not appear in an <u>await</u>) of process S_i. Then for all j, $j \neq i$, T does not interfere with $\{P_i\}$ S_i $\{Q_i\}$."

To make this definition applicable to our programs we may rewrite P5.13 by replacing the atomic actions by corresponding await statements (await true... for the first action in c_1'). Preconditions pre(T) are defined in [88] simply as "assertions preceding T". It seems to me that in the case of P5.13 and (5.26) the clause (3.4b) does not apply and (3.4a) is always satisfied, thus giving the (counterintuitive) result that (5.26a) and (5.26b) are interference-free.

The correct proof of P5.13 should have taken into account the intermediate values of x and y as well. The assertions (5.26) should accordingly be strengthened and an intermediate assertion involving the values of y should be placed between the two statements of c_1'. The above argument serves to show that it is essential for the general applicability of the Owicki-Gries method, not only that auxiliary variables are introduced, but also that all assertions are as strong as permitted by the interference-freeness property. Weaker local assertions will not just (as one could have expected) give rise to weaker global assertions, but may even (as this example shows) encourage incorrect deductions.

It would of course be interesting to have at hand a statement to the effect that for every program c one can introduce auxiliary variables and assertions with the above characteristics between atomic actions, such that these assertions form a proof for the statement m(c). I conjecture that this is true and that it should not be too difficult to prove. We would then know that there are two different but equivalent ways of looking at the same semantics.

The auxiliary variables so defined tend to express the control points within sequential components. Thus control is introduced in a roundabout way, sometimes leading to circumstantial proofs of simple facts. Since, it seems, control has to be taken into account in some way or other, I think it is just as well to do so in the first place, which is what I consider to have been achieved by the control sequence formalism. Thus, I think this work can also be seen as putting the Owicki-Gries method into the perspective of being but one method for proving concurrent programs. (I would, on this account, disagree with

Andrews who seems to imply in [2] that the Owicki-Gries method underlies all proofs of concurrent programs.)

In writing a proof $\{P_1\}c_1\{Q_1\}$ we implicitly set up a relation between the initial states and the final states of c_1. I would argue that c_1 may not be an appropriate substructure of c at which to apply relational reasoning. This is manifestly true for P5.13: the fact that c_1', by itself, transforms $x=x'$ into $x=x'+2$ is of no significance whatsoever in its proof. And in particular, the relational meaning (that is, the "strongest proof") of a general c cannot be given from the relational meaning of its constituent c_j only.

With the semantics given in section 5.2.2, I have proposed to view the atomic actions in c as (by definition) accurately those portions of c whose relational semantics really matters. This is why in this thesis the semantics of c has been defined in terms of the $m(a_i)$ rather than the $m(c_j)$.

In the "control sequence" approach the usual input/output proof rules for <u>if</u> clauses and loops ([30] and sections 2.2 and 2.3) break down. This is however only to be expected. As the discussion relating to program P5.8 in section 5.2.2 shows, we cannot expect that in an <u>if</u> clause which is not itself an atomic action, the usual relations hold between its input and output states. The control sequences of a sequential component, by capturing all of its "control" but none of its "state", represent precisely those aspects of that component which are invariant which respect to interference by other components. Our additional condition (5.11) allows for interference unless the state is transformed "atomically".

Perhaps it should be added that our semantic formalism does not, of course, preclude proofs of concurrent programs being conducted as "statically" as possible. Indeed, I think that one should always (whichever technique is used) strive for ways of avoiding to have to consider all possible execution sequences one by one. But I claim that because control is taken into account explicitly and because relational semantics is applied where it matters (namely for the atomic actions of c), the approach taken here is convenient for the purposes of defining the

semantics of the programs under consideration. Just the fact that we use the word "execution" does not warrant any charge that the formalism advanced in section 5.2.2 is "unmathematical" or "implementation-dependent"; everything has been defined purely in terms of the program text.

Because of the present separation between semantics and correctness criteria, our formalism appears to give little general insight into how correctness can best be established in every individual case. However, even though a methodology for proving concurrent programs correct has not been our immediate concern, a "proof method" can be associated with the control sequence formalism. This consists, not of enumerating all execution sequences, but of examining the properties of one or a few general execution sequences. A proof conducted in this manner is described in section 6.2.3.

Summary of section 5.2.5

We relate our semantic formalism to the Owicki-Gries method of proving parallel programs and we expose some difficulties in the latter. We show that their method requires, in general, both the introduction of auxiliary variables and the consideration of the strongest possible intermediate assertions. We show that this is a consequence of the fact that the sequential components of a concurrent program may not be appropriate substructures for the application of relational reasoning.

5.3 Relational Characterisation of Atomic Actions

5.3.1 Introductory Remarks

The previous section 5.2 has been devoted to the syntactic and semantic definition of the class of programs which are the main subject of this thesis. We have, in particular, defined the parallel operator || in terms of the relational semantics of the atomic actions used in

connection with it. Thereby we have also, implicitly, defined the semantic properties of those atomic actions. The present section is intended to work out these properties more explicitly, and to establish a relation to the atomicity criteria defined in chapter 4.

Accordingly, it has two parts. In the first part (subsection 5.3.2) a property of atomic actions will be identified which I call "effect-replaceability". We shall argue with the help of two propositions (5.3 and 5.4) that effect-replaceability is the characteristic "static" semantic property of atomic actions. We then discuss effect-replaceability in connection with the invariant-assertion proof method for concurrent programs (proposition 5.5). In the second part (subsection 5.3.3) we establish a link between the "static" atomicity criteria discussed in the present chapter and the "dynamic" criteria defined in chapter 4. In particular, with propositions 5.6 and 5.7 we establish an equivalence of sorts between the two criteria.

5.3.2 Effect-Replaceability

In order to avoid talking about the proper nesting of atomic actions all the time we concentrate on the set of atomic actions on a particular level of nesting and their enclosing actions. We can safely do so because of the general recursion property associated with rule (SYN3e) in section 5.2.1. Thus, let a concurrent program c and one of its atomic actions a_1 be given. We denote by $encl(a_1)$ either the atomic action immediately enclosing a_1, or (if a_1 is already an outermost atomic action) c itself. Further, we call two atomic actions a_1 and a_2 "on the same level of nesting" iff $encl(a_1)=encl(a_2)$.

Henceforth we concentrate on a set of atomic actions $a_1, a_2, \ldots$ all of which are on the same level of nesting, together with their enclosing action. To be explicit, we can take all outermost atomic actions and their enclosing action c. The discussion in this subsection is conducted under the assumption that all of the a_1 are actual programs, i.e. syntactically derivable from PROG. However, our discussion can easily be extended for atomic conditions or atomic guarded commands.

Because in our definition (5.16) of the semantics of c only the $m(a_i)$ with $c=encl(a_i)$ are used, it follows that if any such a_i is replaced by another a' with the same effect relation (i.e. $m(a')=m(a_i)$) then the semantics of c remains unaffected. We state this as

Proposition 5.3 $\forall a'$: $(m(a_i)=m(a') \Rightarrow m(c)=m(c[a_i \leftarrow a']))$

where $c=encl(a_i)$ and $c[a_i \leftarrow a']$ is c in which a_i is textually replaced by a' (outer atomicity brackets are assumed to remain around a').

I call the property described in proposition 5.3 the "effect-replaceability" of an atomic action a_i with respect to its environment c. Further, I claim that effect-replaceability is, in a certain sense, the characteristic property of atomic actions. To see this, let us define effect-replaceability more generally, not just applying to atomic actions. Let therefore d be any textual part of c which is by itself a syntactic entity derivable from PROG; d may not be a single atomic action. We call d "effect-replaceable with respect to c" iff

$$\forall d': (m(d)=m(d') \Rightarrow m(c)=m(c[d \leftarrow d'])) \qquad (5.27)$$

For instance, the assignment "x:=x+1" is effect-replaceable with respect to P5.14 (see below); the same is true for the second assignment "x:=x-2".

```
x:=x+1;
x:=x-2
```

Program P5.14

Proposition 5.3 then simply states that every atomic action satisfies (5.27).

In order to substantiate our claim that (5.27) is the characteristic property of an atomic action we need some kind of converse of proposition 5.3, stating roughly that whenever a program d satisfies (5.27) then it is atomic. This can be inferred from taking the special case

d'=<d> in (5.27). Clearly, m(d)=m(<d>) and therefore, by (5.27), m(c)=m(c[d←-<d>]). We state this as

Proposition 5.4 Let d be effect-replaceable with respect to c;
then m(c)=m(c[d←-<d>]).

Proposition 5.4 states that enclosing any d satisfying (5.27) in atomicity brackets will not alter the semantics of its environment c. Thus d can itself be viewed as an atomic action; for instance, both assignments in P5.14 can be viewed as atomic actions. As atomicity brackets have not actually been put around them, they can be called "unplanned atomic actions" [97].

Caution must be observed when the effect-replaceability property is applied in a circumstance in which atomicity brackets have been omitted due to the local variable rule discussed in section 5.2.3. Let us consider the example

x:=x+1 || <y:=y+1>

Program P5.15

which has (by definition) the same semantics as program P5.16 below.

<x:=x+1> || <y:=y+1>

Program P5.16

Intuitively, we would like to view the first assignment in P5.15, i.e. "x:=x+1", as an "unplanned atomic action" satisfying the effect-replaceability property (5.27). However it doesn't. If we replace "x:=x+1", say (as in [67]), by the effect-equivalent piece of program

<y:=-y>;x:=x+1;<y:=-y> (5.28)

then the resulting program

```
(<y:=-y>;
 x:=x+1;        ||      <y:=y+1>
 <y:=-y>)
```

Program P5.17

differs drastically from both P5.15 and P5.16.

Apparently, we cannot admit (5.28) as a replacement for "x:=x+1" in the context of P5.15. A reasonable way out would be to admit only such pieces of program as a replacement for "x:=x+1" which do not have side-effects on other variables, even if those side-effects are only "temporary". Thus, we would admit

```
x:=x-1;x:=x+2
```

as a replacement for "x:=x+1", but not (5.28). However, this requirement leads us straight to the question of what "other variables" and "side-effects" mean; these "other" variables may not as easily as in our present example be determined syntactically. We are thus bound to enter the same discussion as is going to be conducted in section 5.4.

There is also the following less painful way out. We already know that "x:=x+1" in P5.15 is an "unplanned atomic action" because the rule in section 5.2.3 implies that (by definition) the overall semantics of P5.15 does not change if "x:=x+1" is enclosed within atomicity brackets. Before replacing "x:=x+1" we have to reinsert those brackets. In particular, we are thus allowed to replace "x:=x+1" by the single atomic action

```
<y:=-y;x:=x+1;y:=-y>
```

(as opposed to (5.28)) and we know already (proposition 5.3!) that the resulting program, i.e. P5.18 below, is effect-equivalent with P5.15.

```
<y:=-y;
 x:=x+1;        ||      <y:=y+1>
 y:=-y>
```

Program P5.18

Property (5.27) characterises the "semantic well-behavedness" of atomic actions. According to it, atomic actions are not only syntactic, but also meaningful semantic substructures of a concurrent program. This can be taken advantage of in the proof of a concurrent program using atomic actions by the following frequently applicable proof method for such programs (which has perhaps first been stated in general terms in [33]). In order to prove the invariance of an assertion one has to prove its initial truth and the fact that it is an invariant over all atomic actions contained in the program. Thus, one is in general free to consider atomic actions individually, one by one, thus factoring out a large proof into many different smaller proofs. We shall see this method applied in chapter 6.

Let us state its justification more precisely. To this end, we call a (unary) predicate P: S → {true,false} an "invariant over a program c" iff

$$\forall s,s' \in S: (P(s') \;\&\; (s',s) \in m(c)) \Rightarrow P(s) \qquad (5.29)$$

We then have:

Proposition 5.5 Let c be a program and P an invariant over all a_i with $c=encl(a_i)$.
Then P is an invariant over c.

Proposition 5.5 follows immediately from the fact that, under the assumptions given, the truth of P "propagates" through all execution sequences (5.10).

Again, (5.27) can be seen as the characteristic property of atomic actions which makes proposition 5.5 hold. For effect-replaceability of a_i means that a_i is replaceable, as it were, by a single atomic assignment statement which, applied to a state in which P is true would preserve the truth of P. Any execution can then be viewed as a succession of such assignment statements, clearly leaving P true if P was true initially.

It is perhaps possible to view (5.27) as an instance of a rather more general "semantic well-behavedness" axiom. I have in mind to allow the substitution, in place of the relational semantics m(c), of any other semantics of interest, say "sem(c)". (I am aware - see the final remarks in section 5.2.2 - that the relational semantics is not a "complete" semantics.) We may thus contemplate the following property:

$$\forall d' : (sem(d)=sem(d') \Rightarrow sem(c)=sem(c[d \leftarrow d'])) \qquad (5.30)$$

applicable to syntactically well-defined portions d of c. Property (5.30) would seem to express a general semantic "induction" property, namely that, within the context c, d behaves exactly as it would according to its semantics sem(d) as calculated independently of c, i.e. out of context. In other words, in order to calculate the semantics sem(c) of c one would be able to reason about d in isolation, derive sem(d) and then use only the latter in the calculation of sem(c).

(5.30) can then be specialised to whatever particular semantics one may be interested in; in this way, one may obtain different sorts of semantically well-behaved "portions" of a program. In particular, if sem(.) is specialised to mean the relational semantics m(.) then one obtains the characteristic property (5.27) of atomic actions. This illustrates anew the remark (made, e.g., at the end of section 5.2.2) that relational semantics is just appropriate for atomic actions.

It seems to me that a property very akin to (5.30) can be found several times in other literature. In [84], for instance, Milner uses such a property to define the notion of "behavioural congruence". In this case we have an equivalent of (5.30) where sem(.) is <u>not</u> the

relational semantics. Also, we may quote Habermann from [47], page 70:

> "It must be possible to replace a functional unit without affecting the rest of the system as long as the external specifications of that function remain the same."

Apparently, Habermann here had in mind a property similar to our "semantic well-behavedness" property (5.30).

This completes our discussion of the effect-replaceability property (5.27). If the reader so wishes, he could take (5.27) as a "local" static atomicity criterion (as opposed to the "global" static atomicity criterion (5.16)) and see an analogy to the two dynamic criteria defined in sections 4.2 and 4.3. Propositions 5.3 and 5.4 would then be analogous to the two parts of proposition 4.2. Of course, two different sets of atomicity criteria having been given, there need now to be a few remarks as to their relationship. This is what we turn to in the next subsection.

Summary of section 5.3.2

We identify "effect-replaceability" as a characteristic property of atomic actions. Also we briefly discuss the invariant-assertion proof method for our programs.

5.3.3 Relationship Between Static and Dynamic Atomicity Criteria

This section contains some thoughts on the relationship between our various atomicity criteria defined in chapters 4 and 5. In chapter 4 we have discussed a pair of what I have called "dynamic" atomicity criteria. The reason for calling these criteria "dynamic" was that our basic objects of interest were executions, represented in the form of occurrence graphs. Our first dynamic criterion (defined in section 4.2) has been called "global" because it expresses a property of an execution as a whole. The second dynamic criterion (defined in section 4.3) has been called "local" because it expresses a property of those parts of a

structured occurrence graph which correspond to atomic action executions. The relationship between these two criteria has been given by proposition 4.2.

Similarly, in the present chapter we have discussed a pair of what might be called "static" atomicity criteria. The reason for calling these criteria "static" is that our basic objects of interest are programs rather than executions. Again, we have defined a criterion (namely, (5.16)) which could be called "global" because it expresses the semantic properties of atomic actions implicitly in terms of the semantics of the enclosing program. We have then defined a corresponding "local" static criterion (5.27) which has been so called because it expresses the semantics of atomic actions directly rather than implicitly. The relationship between these two static criteria has been given by propositions 5.3 and 5.4.

The objective of this section is to relate these pairs of static and dynamic criteria to each other. Ideally, one would like to have at hand a set of propositions showing the equivalence (or essential equivalence) of the respective global and local criteria. We shall give such propositions in some detail only for the two global criteria. The relationship between the two local criteria will be discussed in general terms only.

Recall that the global dynamic criterion (section 4.2) postulates the cycle-freeness of the structured occurrence graph describing an execution. The global static criterion (5.16a), on the other hand, postulates the existence of a serial execution leading from an initial state s' to a final state s in order for (s',s) to be in $m(c)$. In the interest of relating these two criteria to each other, we first of all have to get a clash of terminology out of the way. So far, we have reserved the word "execution" for two different purposes: both to denote a (structured) occurrence graph and to denote a sequence of the form (5.10). We shall see below that the latter can meaningfully be viewed as a specially inscripted occurrence graph, so that no terminological confusion will arise.

Our aim is to establish the following connection between these two criteria. We shall first show (proposition 5.6 below) that if an execution of c, described by a cycle-free structured occurrence graph, leads from an initial state s' to a final state s then $(s',s) \in m(c)$. As discussed in section 4.5, the immediate converse of this result does not in general hold true. However we will show the following weak converse (proposition 5.7 below): given a cyclic structured occurrence graph then one can construct a program c such that this structured occurrence graph can be viewed as an execution of c leading from an initial state s' to a final state s but $(s',s) \notin m(c)$.

Intuitively, these two propositions are obvious enough. The main difficulty in proving them lies in actually drawing up a correspondence between our two underlying models, i.e. the occurrence graph model of chapter 4 and the control sequence formalism of section 5.2. We need the notion of a structured occurrence graph describing a "valid and complete" execution of a program c. Our plan is to define this notion in analogy to the examples discussed in chapter 4. The "trick" for relating occurrence graphs and programs will be inscriptions on the events of the graph which give information about the actions in the program whose executions these events represent.

We illustrate this definition by the following example.

$$\langle\langle x:=x+1\rangle \;||\; \langle x:=2*x\rangle\rangle \;\;||\;\; \langle x:=x-1\rangle$$

Program P5.19

Let us abbreviate the first atomic action in P5.19 by "a_1", i.e.

$$a_1 \;=\; \langle\langle x:=x+1\rangle \;||\; \langle x:=2*x\rangle\rangle$$

Consider the following execution of P5.19 which starts with x=1:

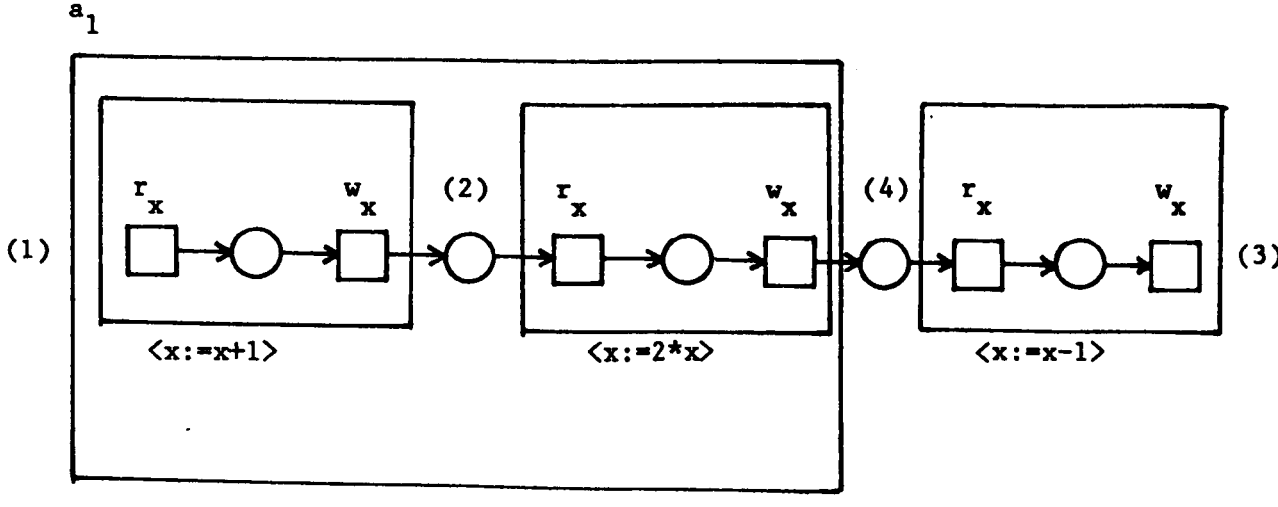
$$(x=1)a_1(x=4)\langle x:=x-1\rangle(x=3) \qquad (5.31)$$

(5.31) shows that $(x=1,x=3) \in m(P5.19)$. The fact that $(x=1,x=4) \in m(a_1)$

which is essential (5.11!) for the validity of (5.31) can be inferred from the existence of the following sequence:

$$(x=1)\langle x:=x+1\rangle(x=2)\langle x:=2*x\rangle(x=4) \qquad (5.32)$$

Each of the two assignments in (5.32) can, in analogy to the examples in chapter 4, be further decomposed into a read event on x, say "r_x", followed by a write event on x, say "w_x". In total, (5.31) could be decomposed in the way shown in Figure 5.1.



Note: "(x=j)" has been abbreviated to "(j)"

Figure 5.1: (G_1,T_1)

This decomposition need of course not be unique. However (as in section 4.1) we can assume that there always exists such a decomposition down to the level of single reads and writes.

The graph (G_1,T_1) shown in Figure 5.1 satisfies all of the axioms of a "structured occurrence graph" (section 3.3), except (unimportantly) the requirement that there is a single outermost atomic activity. The activities and the conditions of the graph bear inscriptions showing which actions of the program P5.19 are "presently" being executed and, respectively, which state the system is "presently" in.

Without these inscriptions there could be no way of deciding which state transformation has actually been effected by the execution described by the graph. Indeed, the graph (G_2,T_2) shown in Figure 5.2 below describes the same basic events in the same order as (G_1,T_1).

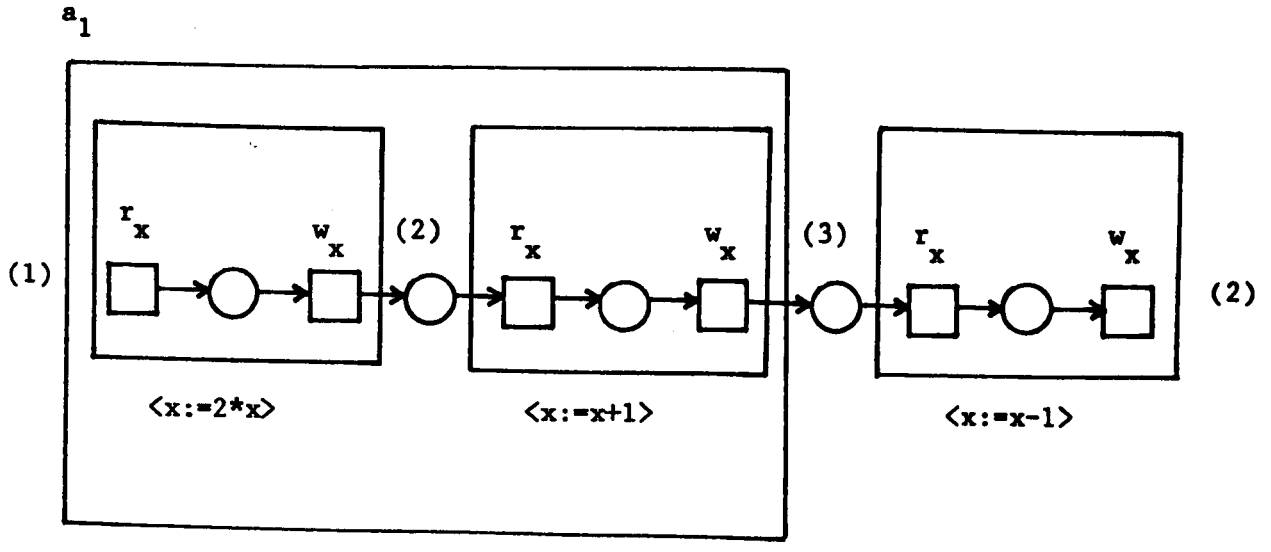


Figure 5.2: (G_2,T_2)

However, due to the reversed order of the occurrence of the first two actions, (G_2,T_2) transforms (x=1) into (x=2) rather than (x=3).

Of course, all operations defined in section 3.3 can be applied to these graphs as well, in particular the "collapsing" operation. For example, (G_1,T_1) can be collapsed to give

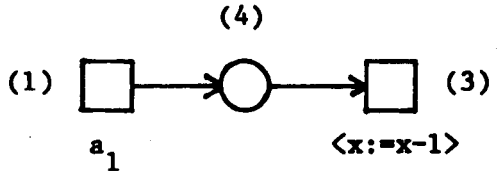


Figure 5.3

which represents the sequence (5.31) as an inscripted occurrence graph. Similarly, (G_2,T_2) can be collapsed to give

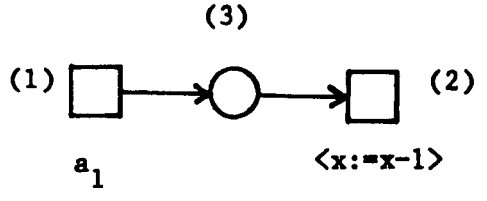


Figure 5.4

which is a description of the following valid (and complete) execution:

$$(x=1)a_1(x=3)\langle x:=x-1\rangle(x=2) \tag{5.33}$$

Note that the graphs shown in Figures 5.3 and 5.4 can again only be distinguished by their inscriptions.

Thus we have seen how every execution sequence (5.10) can be represented by an inscripted occurrence graph. If desired the latter can be refined down to the detail of actual reading and writing of variables; thus one may obtain an (inscripted) structured occurrence graph (G,T). If $u = s_0a_1\ldots a_ns_n$ is the execution sequence in question then we write

$$s_0\ (G,T)\ s_n \tag{5.34}$$

in order to express the fact that the structured occurrence graph (G,T) constructed as above transforms the initial state s_0 into the final state s_n. For example, we have

$$(x=1)\ (G_1,T_1)\ (x=3) \quad \text{and} \quad (x=1)\ (G_2,T_2)\ (x=2),$$

respectively, for the structured occurrence graphs shown in Figures 5.1 and 5.2.

We do not interpret (5.34) as expressing a "relation" between initial states and final states. (G,T) represents only a single execution from s_0 to s_n. It may so happen that from another initial state there does not exist any execution which could be described by (G,T). Furthermore, we assume the final state s_n to be uniquely determined by the initial state s_0 and the details of the inscripted graph (G,T).

We now generalise and consider any arbitrary (inscripted) structured occurrence graph (G,T). Our objective is to define the conditions for (G,T) to be a description of an execution of a given program c starting in a given initial state s´. We require the basic events of (G,T) to be read or write events to variables of c, according to the rule that the Boolean expressions in c generate a number of read events and the assignments V:=E of c generate a number of read events to variables in E followed by a single write event to V. We require the following three additional conditions to hold for (G,T):

(i) No two write/write or read/write events to the same variable are concurrent.

(ii) The basic graph G (is acyclic and) describes a complete execution of c, starting in s´.

(iii) The activities in T correspond to atomic actions in c (and can therefore be inscripted in the same way as illustrated in the above example graphs).

We do not go into any formal detail in defining these conditions. (i)

and (iii) should be self-explanatory. (ii) can be defined analogously to the propeties of "validity" and "completeness" defined in section 5.2. We require that when only those parts of (G,T) which correspond to a given sequential component are considered then the inscriptions on this part describe a valid and complete execution of this component. The completeness condition implies in particular that for loops

$$\underline{\text{do}}\ B \rightarrow c\ \underline{\text{od}},$$

the last event(s) generated is (are) read events on the variable(s) contained in B, determining the ultimate falsehood of B.

The requirement that (G,T) be so refined as to contain the read and write events to individual variables of c ensures that the initial state s´ together with (G,T) and its inscriptions determine uniquely all intermediate states as well as the final state s holding after the execution described by (G,T). Again, we may express this by writing

$$s'\ (G,T)\ s, \tag{5.35}$$

thus generalising (5.34). Because of their association with execution sequences u, the structured occurrence graphs under consideration in (5.34) are linearly ordered on all levels of abstraction. In (5.35), by contrast, the only ordering requirements imposed on (G,T) are (i) and (ii) above.

Thus we may, for instance, consider the following graph which satisfies (i)-(iii) for P5.19.

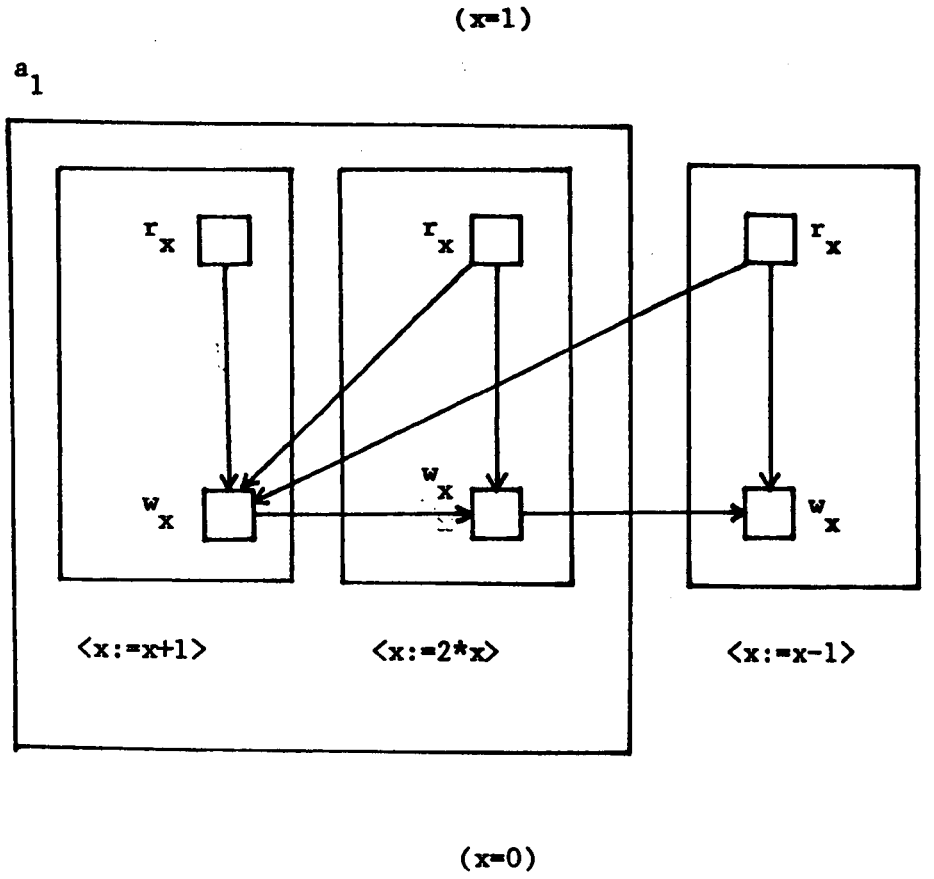


Figure 5.5: (G_3,T_3)

(G_3,T_3) satisfies the properties of a complete execution of P5.19 because all of its basic read and write events occur in an order satisfying the individual requirements of the assignments contained in P5.19. On the other hand, (G_3,T_3) does not "satisfy atomicity" in the sense of section 4.2, and sure enough it transforms (x=1) into (x=0) which is not in (x=1)m(P5.19).

We can now state our first result more formally as follows. Let c be a program and s´ an initial state. Let (G,T) moreover be an inscripted occurrence graph satisfying (i)-(iii) for c and s´. Then

Proposition 5.6 If (G,T) satisfies atomicity and s´ (G,T) s
then (s´,s) ∈ m(c).

The proof of proposition 5.6 may make use of proposition 4.1. Because (G,T) satisfies atomicity it can be serialised so that it describes a linear order on all levels of abstraction. This holds in particular for the outermost level. But the inscriptions on the events of this level, together with the (uniquely determined) intermediate states, can then be arranged in an execution sequence (5.10) from s to s´ which is complete because (G,T) satisfies (ii). Its validity follows from considering the graph recursively down to lower levels of abstraction. Hence (s´,s) $\in$ m(c).

As already stated several times, the immediate converse of proposition 5.6 does not necessarily hold. To see this again on the present example, we may consider the structured occurrence graph shown in Figure 5.6 below which violates atomicity but nevertheless transforms (x=1) into (x=2), and (x=2) $\in$ (x=1)m(P5.19).

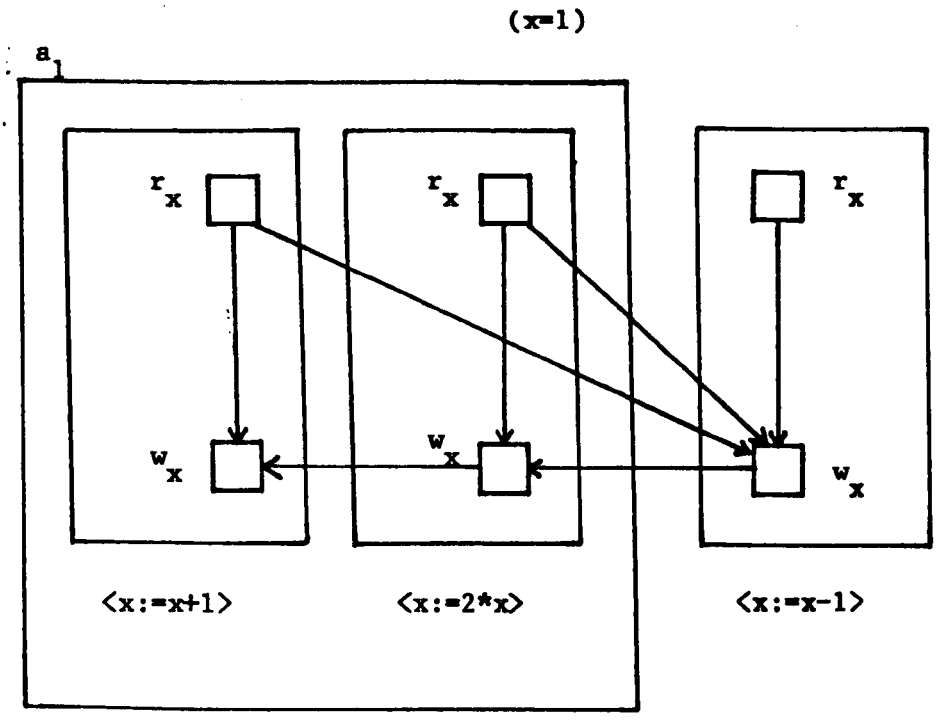


Figure 5.6

However, for every atomicity-violating structured occurrence graph we can easily construct a program c, an initial state s´ and suitable inscriptions, such that the resulting inscripted graph describes a complete execution of c leading to a final state not in s´m(c).

We state this more formally as

Proposition 5.7 If (G,T) violates atomicity
then $\exists$ c,s´,s: (s´ (G,T) s & (s´,s) $\notin$ m(c)).

We will not give the general proof of proposition 5.7. It consists essentially of a generalisation of the following argument. Suppose there is a cycle between two activities A_1 and A_2 of (G,T). Then A_i must contain events e_i and f_i (i=1,2) with $e_1 < f_2$ and $e_2 < f_1$ (see Figure

5.7).

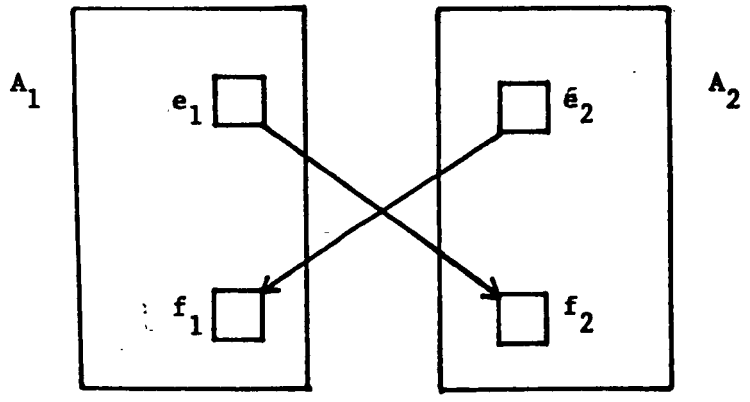


Figure 5.7: (G_4,T_4)

Of course, $e_1,f_1 \notin \{e_2,f_2\}$ and $e_2,f_2 \notin \{e_1,f_1\}$ because T_4 is well-nested; however, possibly $e_i=f_i$ $(i=1,2)$. Depending on whether $e_i \underline{<} f_i$, $f_i \underline{<} e_i$ or e_i co f_i $(i=1,2)$, we can take the e_i and f_i to be the read/write events to suitably defined variables. For example, in Figure 5.7 where e_i co f_i $(i=1,2)$ we may decide that e_1 and e_2 should be writes to variables x and y, respectively, and that f_1 and f_2 should be reads to y and x, respectively. We introduce two auxiliary variables z1 and z2 and consider the program

$$\langle x:=1 \;||\; z1:=y\rangle \;||\; \langle y:=1 \;||\; z2:=x\rangle$$

Program P5.20

Abbreviating (x,y,z1,z2)=(i,j,k,l) to (i,j,k,l), we see that

$$(0,0,0,0)m(P5.20) = \{(1,1,0,1),(1,1,1,0)\}$$

but that, with the inscription e_1: x:=1, f_1: z1:=y, e_2: y:=1 and f_2: z2:=x, we have

$$(0,0,0,0)\ (G_4,T_4)\ (1,1,1,1).$$

Taken together, propositions 5.6 and 5.7 establish an equivalence of sorts between the global dynamic atomicity criterion defined in section 4.2 and the global static criterion (5.16a) defined in section 5.2.2. In practical terms, propositions 5.6 and 5.7 signify the following. Proposition 5.6 states that even though the semantics of a concurrent program c has been phrased in terms of linear sequences, any particular implementation of c does not necessarily have to actually sequentialise (i.e. mutually exclude in time) the atomic actions contained in c. Rather, any (concurrent) execution of c is acceptable if it satisfies atomicity in the sense of section 4.2. Proposition 5.7 states that any implementation of c which admits atomicity-violating executions runs the risk of producing final states outside the effect relation of c, i.e. incorrect results. In order to avoid this, more semantic knowledge about the program would be required. A similar conclusion has already been arrived at in section 4.5.

For aesthetic reasons, it would be nice if an analogue of the last two propositions could be found for the two local atomicity criteria. We would then be in possession of a full set of correspondences between our four criteria. I have however not tried to obtain such a result, mainly because I believe that this question is rather marginal for the present investigation. We are here chiefly interested in whether or not an execution as a whole satisfies atomicity, rather than in pinpointing (if it doesn't) exactly which one(s) of its activities is (are) at fault. This latter question may become important if atomicity is to be implemented by error recovery methods as stipulated, for example, in [16]; it is however beyond the scope of this thesis.

Summary of section 5.3.3

We establish an equivalence of sorts between the global dynamic atomicity criterion (section 4.2) and the global static atomicity criterion (section 5.2.2): (i) An execution which satisfies atomicity effects a desired state transition, and (ii) An execution which violates atomicity effects an undesired state transition for some suitably constructed program.

5.4 Semantic Independence of Actions

In thic section we address ourselves to the following question: what are the conditions for two atomic actions to be "semantically independent"? It should be clear from earlier discussions, say in section 4.5, why this question could be of interest. In particular, if two actions are semantically independent then there exists a fully concurrent implementation (no cross-dependencies in the occurrence graph). We will not give a full answer to this question; rather, the ideas contained in this section are very tentative only. The question of semantic independence is related to questions about information flow in programs, as discussed by Cohen in [26] and Furtek in [40]. A PhD Thesis by Lengaur on this topic is forthcoming [76].

We focus our attention on two concurrent assignments:

$$\langle V_1 := E_1 \rangle \;||\; \langle V_2 := E_2 \rangle \qquad (5.36)$$

where V_1 and V_2 are variables (not necessarily different) and E_1 and E_2 are expressions. We shall examine the semantic independence or otherwise of these two assignments.

We consider some examples in order to illustrate the notion we wish to capture. In P5.21 below we would like to define the two assignments as "not independent" (or "interacting", as the word has been used in section 1.1).

$$\langle X := X+1 \rangle \;||\; \langle X := X+1 \rangle$$

Program P5.21

In this section we use capital letters X,Y,Z for variables.

Programs P5.22-P5.27 below provide examples for what we would like to call "sematically independent" actions.

```
<X:=X+1>  ||  <Y:=Y+1>
```

Program P5.22

```
<X:=Z>  ||  <Y:=Z>
```

Program P5.23

```
<X:=0*Y>  ||  <Y:=1>
```

Program P5.24

```
<X:=Y>  ||  <Y:=Y>
```

Program P5.25

```
var Z: {0,1};
<(X,Y):=(X*Z,Y*(Z⊕1))>  ||  <(X,Y):=(X*(Z⊕1),Y*Z)>
```

Program P5.26

```
var X,Y: {0,1,2,3};
<X:=2*(Y mod 2)+(X mod 2)>  ||  <Y:=2*(X mod 2)+(Y mod 2)>
```

Program P5.27

Let us discuss P5.22-P5.27 in order. The two assignments in P5.22 operate on entirely different variables and are therefore independent.

In P5.23, the common variable is only read but not changed. In P5.24 there is a common variable (namely, Y) which is both read and overwritten but the reading "has no semantic effect". In P5.25, again Y is both read and overwritten but the writing "has no effect".

In P5.26 (where $\oplus$ denotes binary addition) we have two cases. If Z=0 then the left hand assignment overwrites only X while Y is left unchanged, and the right hand assignment overwrites only Y while leaving X intact. If Z=1 it is just the other way round. Thus, although the two actions in P5.26 "look" interacting, they are "really" independent.

P5.26 is an example for two independent assignments to the same variable. This shows that mutual exclusion of writes to the same variable may not be necessary. Thus it may be possible to refine our rule (sections 4.1 and 5.3.3) that write/write accesses to a common variable must be mutually exclusive.

A similar remark is true for P5.27. We can appreciate the semantic independence of the two actions in P5.27 by imagining the two variables X and Y to be split into two bits X_i and Y_i, respectively (i=0,1), so that $X=2*X_1+X_0$ and $Y=2*Y_1+Y_0$ (see Figure 5.8). The left hand assignment in P5.27 reads only Y_0 and overwrites only X_1 while the right hand assignment reads X_0 and overwrites Y_1.

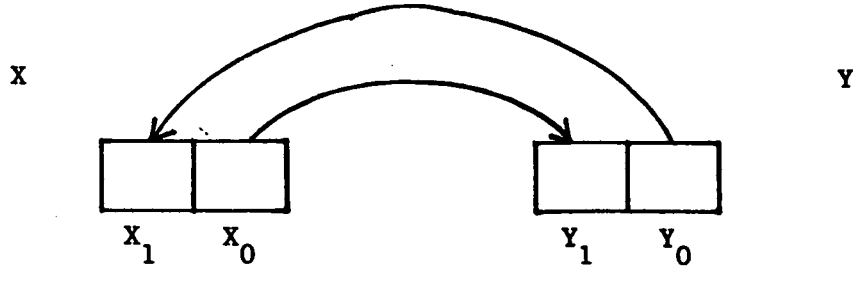


Figure 5.8

Thus, P5.27 could be rewritten as follows:

$$\langle X_1 := Y_0 \rangle \;||\; \langle Y_1 := X_0 \rangle$$

Program P5.27'

In P5.27', the semantic independence of the two assignments is obvious (i.e. syntactically visible). However, since the two assignments are independent even if X and Y cannot be decomposed into bits (say if X and Y have five values, but not if they have only three values initially!) we would like to define semantic independence without recourse to bit decompositions.

Nevertheless, I think that the last example can give us a clue as to how our definition of independence could be approached. If S is the state space of P5.27 then we have two different decompositions of S. One is defined by the programmer as follows:

$$S = X \times Y \tag{5.37}$$

giving 4*4=16 states. The other can be viewed as being "imposed by the assignments in P5.27" as follows:

$$S = X_1 \times X_0 \times Y_1 \times Y_0 \tag{5.38}$$

(again, of course, giving 2*2*2*2=16 states).

From a semantic point of view we are not interested in the structure given to the state space by the programmer (which might be called the "syntactic" or "subjective" structure), but rather we are interested in the structure imposed on the state space by the program itself (which might be called the "semantic" or "objective" structure). It remains to be seen whether or not such a semantic structure, generalising our example P5.27', can always be well-defined.

Let us assume that it can. Thus, let us assume that for every assignment

$$V:=E \tag{5.39}$$

we can always find two variables V^{in} and V^{out}, denoting its "semantic input" and "semantic output", respectively. V^{in} and V^{out} would be required to be such that (5.39) is effect-equivalent to

$$V^{out} := V^{in} \tag{5.40}$$

Perhaps, in order to achieve such a transformation, we would need to generalise the notion of a "variable".

Suppose, for the moment, that we can always arrange for a general transformation of (5.39) into (5.40). The "semantic independence" between two different assignments $\langle V_1 := E_1 \rangle$ and $\langle V_2 := E_2 \rangle$ could then be defined as follows. Given the two assignments, calculate the four "variables" V_1^{out}, V_1^{in}, V_2^{out} and V_2^{in}, and check whether the following three conditions hold:

(i) V_1^{out} and V_2^{out} are independent;

(ii) V_1^{out} and V_2^{in} are independent;

(iii) V_2^{out} and V_1^{in} are independent.

Thus we reduce the independence of actions to the independence of "variables" to be defined. The independence of variables is a better understood notion; we can call two variables V_1 and V_2 "independent" if by fixing a value of V_1 one does not also at the same time fix part of a value of V_2 and vice versa. Thus the three variables X, Y and Z would be mutually independent, but the two variables (X,Z) and (Y,Z), though different, would not be independent (unless Z has only one possible value).

The rationale for (i)-(iii) consists in a consideration of the following two assignments:

$$\langle V_1^{out} := V_1^{in} \rangle \;||\; \langle V_2^{out} := V_2^{in} \rangle$$

which are by assumption equivalent to our initial two assignments (5.36). (i) expresses the absence of overlapping output; (ii) represents the absence of information transfer from the second assignment to the first assignment, and vice versa for (iii). For P5.27, $V_1^{out}=X_1$, $V_1^{in}=Y_0$, $V_2^{out}=Y_1$, $V_2^{in}=X_0$, and (i)-(iii) are satisfied.

This definition of "semantic independence" should satisfy the following two properties:

Property 5.1 If V_1 and V_2 are independent variables and neither V_1 occurs in E_2 nor V_2 occurs in E_1
then $V_1:=E_1$ and $V_2:=E_2$ are semantically independent.

Property 5.2 If $V_1:=E_1$ and $V_2:=E_2$ are semantically independent then
$m(V_1:=E_1;V_2:=E_2) = m(V_2:=E_2;V_1:=E_1)$
(i.e. the two assignments commute, or satisfy the Church-Rosser property).

As seen on our examples, neither the opposite of property 5.1 nor the opposite of property 5.2 need to hold. P5.24 shows a program whose actions are independent but V_2 occurs in E_1; P5.21 shows an example in which the actions commute but are not independent.

The above have been stated as "properties" rather than "propositions" because, of course, they cannot be proved yet. The missing link is the definition of the two "semantic variables" V^{in} and V^{out} in the transformation of (5.39) into (5.40). In the remainder of this section I describe the way in which (I would imagine) this concept can be properly captured.

The state space of P5.27 has 16 elements and can therefore easily be dissected into four bits X_0 etc., showing the independence between the two actions. But even if X and Y are, say, five-value variables initially (giving a 25-value state space) then the two actions are independent. In this case we cannot so easily find a decomposition into bits to show the independence.

Therefore we need to generalise. I think the key concept could be that of the "transformation" of variables. Even in case

var X,Y: {0,1,2,3,4} (5.41)

initially, we can define two variables X_1 and X_0 for X (and similarly,

Y_1 and Y_0 for Y) such that the transformation equations

$$X = 2*X_1+X_0 \quad \text{and} \quad Y = 2*Y_1+Y_0 \tag{5.42}$$

hold. For this we may use the following correspondence:

X	X_1	X_0
0	0	0
1	0	1
2	1	0
3	1	1
4	2	0

Figure 5.9

For a 1-1 correspondence between X and the pair (X_1,X_0) we need the further rule that $X_1=2$ implies $X_0=0$. In all, we define

$$X_0 \in \{0,1\},\ X_1 \in \{0,1,2\} \text{ and } (X_1 \neq 2 \vee X_0=0) \tag{5.43}$$

such that (5.42) holds. Correspondingly, we define Y_1 and Y_0 for Y.

With these four variables, P5.27 under the initial condition (5.41) can again be transformed into the program

$$\langle X_1:=Y_0\rangle \ \|\ \langle Y_1:=X_0\rangle \tag{5.44}$$

which shows the independence between the two actions. (i) holds because the variables X_1 and Y_1 are independent. Because $Y_0 \neq 2$ by (5.43), the assignment of Y_0 to X_1 cannot destroy the truth of (5.43), whatever the value of X_0. This shows the independence of X_1 and X_0 in (5.44); similarly, Y_0 and Y_1 are independent in (5.44), which establishes, respectively, (ii) and (iii).

We finally show how variable transformations can be used to establish the independence of the two actions in program P5.26, reproduced

below as P5.28 with X and Y restricted to be binary variables.

```
var X,Y,Z: {0,1};

<(X,Y):=(X*Z,Y*(Z⊕1))>  ||  <(X,Y):=(X*(Z⊕1),Y*Z)>
```

Program P5.28

In the left hand assignment of P5.28, if Z=0 then X is output and if Z=1 then Y is output. It therefore stands to reason to introduce a new variable W_1 as follows:

$$W_1 = X*(Z⊕1) + Y*Z, \tag{5.45a}$$

and similarly a variable

$$W_2 = X*Z + Y*(Z⊕1) \tag{5.45b}$$

for the right hand assignment in P5.28.

With this convention, P5.28 can simply be transformed into

$$<W_1:=0> \;||\; <W_2:=0>$$

Program P5.28'

We have to show that W_1 and W_2 are independent variables. This can be seen by considering Figure 5.10 which is the truth table of the transformation (5.45), showing that there is a 1-1 correspondence between the values of the variable (X,Y,Z) and the values of the variable (W_1,W_2,Z).

X	Y	Z	W_1	W_2
0	0	0	0	0
0	1	0	0	1
1	0	0	1	0
1	1	0	1	1
0	0	1	0	0
0	1	1	1	0
1	0	1	0	1
1	1	1	1	1

Figure 5.10

Hence by choosing a value for W_1 we retain the freedom to choose any value of W_2, and vice versa. Thus (i)-(iii) are satisfied for P5.28', whence the two assignments in P5.28 are independent.

I think that this method of variable transformation could be generally exploited in order to give a precise definition of the notion of independence which we are looking for. However, I have not been able to consider this any further. Variable transformations have occurred also in [30] in the design of a small program.

Summary of section 5.4

We have shown on a few examples what is meant by "semantic independence", and we have shown on these examples how independence could be proved. The proofs involve transformations of the program's variables into new but semantically "more transparent" variables. It was left open whether such transformations can be generalised.

5.5 Possible Syntactic Extensions

Because semantics and proof methods have been discussed elsewhere, I wish in this short discussion section to concentrate on possible

syntactic extensions of the notation described in this chapter. Apart from introducing arrays, records etc., on which I will not comment, there are also a few other conceivable ways in which atomic actions could be allowed to enter the language.

First an (admittedly very contrived) case could be made for allowing the proper overlapping of atomic actions. Consider P5.29 below

$$\langle_1 x:=x-1 \;||\; \langle_2 \; y:=y-1\rangle_1 \;||\; z:=z+1\rangle_2$$

Program P5.29

Suppose, for the sake of illustration, that there is a bank customer who maintains a current account called "y" in some town, a deposit account "z" in the same town and a duplicate current account "x" in another town. Suppose the customer wants to withdraw one unit from his current account y and deposit it on his deposit account z. The above program "expresses" that the withdrawal and the deposit must occur atomically, as must the withdrawal and the simultaneous withdrawal at his duplicate account x; no other requirements are specified.

It is doubtful whether such a program could be defined to make semantic sense. Even if it could, it would be even more doubtful whether the program would make practical sense. Be that as it may, we have excluded such programs, and I see no reason why to allow them.

Another situation we have so far excluded is that in which an atomic action overlaps with a parallel operator ||. This is even more awkward to express syntactically, but consider P5.30 below.

$$\begin{array}{l|l} \langle x:=1\rangle; & \langle y:=1\rangle; \\ \langle x:=x+1 & y:=y+1\rangle \end{array}$$

Program P5.30

P5.30 describes two "processes" consisting of two parallel atomic initialisations, followed by a "common atomic action". Again, we have

excluded this situation; whether it might in the end be necessary to extend our language in this way is open to doubt. As we do not use these extensions, this section serves no purpose other than illustrating the scope of our syntax.

6. CASE STUDIES IN THE DESIGN AND VERIFICATION OF CONCURRENT PROGRAMS

6.1 Introductory Remarks

Having looked at concurrent programs using atomic actions from various angles in the preceding sections, and having examined a number of trivial example programs, in this chapter we now turn to what I would consider to be non-trivial application examples. We consider three different problems and their solutions, each time making specific points along the lines which make a connection to previous chapters. Although I would have liked to stress the actual design aspects of these programs, for example by working out a set of design heuristics, I have not in all cases succeeded in doing so. However, we do concentrate on proving in detail the correctness of our main solutions.

In section 6.2 we reexamine a program known from [33]. This is a short but surprisingly complicated program whose proof (of partial correctness) requires razor-sharp reasoning. We shall give two separate proofs for this program: one applying the invariant-assertion method and another one using the control sequence semantics directly. The latter proof is less elegant than the former, but elegance is not the reason why I describe it. I wish to make the point that indeed control sequence proofs are possible (even for relatively complicated programs) with the same degree of formality as assertional proofs; in particular, I show that we do not have to take into account an overwhelming number of different executions, but rather that we can perfectly satisfactorily work with only one "general" sequence.

In section 6.3 we describe the design of a concurrent program finding Euler cycles in a graph. We give two concurrent solutions to this problem, one of which coincides with the one described in [10]. The other solution is an extension which differs from the first solution both in efficiency and apparent ease of proof. We prove the correctness of the second solution using a mixture of invariant assertions and control sequence type arguments. Again this serves to show that such "mixed" reasoning can be perfectly rigorous; in this case, the further point can be made that it seems to be convenient to include operational

arguments in the proof.

Our last example (to be discussed in section 6.4) concerns a small distributed algorithm which has been described by Chang and Roberts in [25]. We extend this algorithm to work also in the case of simple kinds of failures, and we prove the correctness of the extended algorithm. Apart from this, a more general point made in section 6.4 concerns the translation of programs using buffered communication into programs using shared data; I claim that such a translation is always possible, even for "handshake" communication.

6.2 A Concurrent Fixpoint Program

6.2.1 Introduction

When studying Dijkstra's proof of the fixpoint program described in [33] I found his arguments at first extremely difficult to understand. Naturally, I wondered whether this was because of the inherent complexity of the proof or just because of my not being acquainted with such proofs. Consequently I tried either to find a simpler proof or else to convince myself that Dijkstra's proof is indeed the simplest possible.

As a result of this study, I am now satisfied that Dijkstra's argumentation is indeed the simplest and most elegant way of proving the program. In particular, I have found a different way of deriving his proof, which I believe to be as stringent as possible. This involves the weakening of predicates in a way which is rather analogous to the introduction of auxiliary functions in the solution of a differential equation. Dijkstra's proof and this derivation are discussed in section 6.2.2.

In section 6.2.3 we discuss a different proof of the same program. This represents a first attempt (by me) to apply the control sequence semantics rigorously in the proof of a serious program. I should make it clear that I do not consider this second proof to be any "better" than the first one; in fact it is much less elegant. The major point I wish to make is that proofs taking into accoumt execution sequences can

be conducted with the same degree of rigour and effectiveness as proofs by the inductive assertion method. This point is discussed in more detail in section 6.2.4 where we compare the two proofs.

The following problem is to be considered. Let y be an integer vector with components y_1, ..., y_N ($N \geq 1$) and let f(y) be a vector-valued function of vectors with component functions $f_1(y)$, ..., $f_N(y)$. Throughout the text, we shall use indices i,j,k ranging from 1 to N. Starting with some initial value of y, we wish to write a program which performs a series of atomic assignments

$$\langle y_i := f_i(y) \rangle \tag{6.1}$$

with the objective of establishing the fixpoint relation y=f(y), or, in component equations:

$$\forall i: y_i = f_i(y) \tag{6.2}$$

We shall use the abbreviation "eq_i" to denote the equation "$y_i=f_i(y)$" throughout.

The program to be considered will consist of a parallel combination of N sequential components c_i (see P6.1 below), each of which is responsible for one of the N assignments (6.1). For the purposes of termination, a Boolean vector "h" is introduced, the initial values of whose N components are "true".

```
c_i:   10   do <∃j: h_j> →
       20      if <eq_i → h_i:=false>
       30      [] <not eq_i → y_i:=f_i(y)>;
       40j                    ∀j: <h_j:=true>
               fi
           od
```

Program P6.1

Line 40j is an abbreviation for N assignments the order of which is

unimportant.

The initial value of y and the function f may be such that no fixpoint may be produced by a series of assignments (6.1). However it is asserted that the program is partially correct, i.e. if and when it terminates then y is a fixpoint. Formally, we wish to prove that the formula

$$\exists j: h_j \vee \forall i: eq_i \tag{6.3}$$

is an invariant.

A straightforward attempt to prove the invariance of (6.3) reveals that this is not directly possible, the main obstacle being the atomic action in the first alternative, i.e. line 20, of P6.1. We therefore have to examine the problem more closely. One observation about the proof can be made straight away. Because the assignment in line 30 may render all equalities false, the fact is essential that this possibility is signalled to <u>all</u> components c_j in line 40j. This fact must therefore crucially enter the proof.

If the h_j were set to true within the same atomic action as the assignment to y_i, as in P6.2 below, then we would easily be able to prove the invariant

$$\forall i: eq_i \vee h_i \tag{6.4}$$

which, being much stronger than (6.3), would be more than sufficient for our purposes.

```
c_i:  10   do <∃j: h_j> →
      20      if <eq_i → h_i:=false>
      30      [] <not eq_i → y_i:=f_i(y);
      40j                    ∀j: h_j:=true>
              fi
           od
```

Program P6.2

Unfortunately, however, (6.4) is not invariant over P6.1. Thus, the proof of (6.3) over P6.1 can be expected to elaborate on the difference between P6.2/(6.4) and P6.1/(6.3), using only the fact that all h's are set to true in line 40j.

6.2.2 Derivation of Dijkstra's Proof

We represent the fact that all h's are set to true in line 40j in terms of auxiliary variables. For each component c_i a Boolean N-vector s_{ij} is introduced which is "true" initially. The s_{ij} are set to false by the second alternative in line 30 and are reset to true one at a time by line 40j, as in program P6.3 below.

```
c_i:  10   do <∃j: h_j> →
      20      if <eq_i → h_i:=false>
      30      [] <not eq_i → y_i:=f_i(y); ∀j: s_ij:=false>
      40j                    ∀j: <h_j:=true; s_ij:=true>
              fi
           od
```

Program P6.3

Thus, s_{ij}=false can intuitively be appreciated as stating that "the component c_i may have invalidated the j'th equality but has not yet reset h_j to true". Also, the s_{ij} can be viewed as "control point variables" in the sense of section 5.2.5, s_{ij}=false stating that "c_i is between

line 30 and line 40j".

Let us now attack the proof of (6.3) over P6.3. As we have seen, (6.4) is not an invariant. However, we could try to take (6.4) to be a crude first approximation and try to "make it correct" by "adding" terms which cover the failing cases of (6.4). Thus we arrive at the improved approximation

$$\forall i: (eq_i \vee h_i \vee P_i) \tag{6.5}$$

where the P_i are supposed to express the cases for which (6.4) fails.

The P_i must express that "some component, say the k'th, interferes by having executed its second alternative but not yet set h_i to true". Thus our next guess immediately becomes:

$$\forall i: (eq_i \vee h_i \vee \exists k: \underline{not}\ s_{ki}) \tag{6.6}$$

Indeed, (6.6) is an invariant. It is trivially invariant over the first alternative, line 20. Further, after the second alternative (line 30) the third term of (6.6) always holds. Finally, (6.6) is clearly invariant over line 40j. But unfortunately, (6.6) is too weak for our purposes because it does not imply (6.3)!

This leads us to consider whether it would be possible at all to strengthen (6.6) by conjoining a factor involving the h's to the third term of (6.6), thus cancelling this term when all h's are false. The most plausible way of doing so I can think of is the following:

$$\forall i: (eq_i \vee h_i \vee (\exists k: (h_k \,\&\, \underline{not}\ s_{ki}))) \tag{6.7}$$

Indeed, (6.7) implies (6.3). But unfortunately, (6.7) is not an invariant at all! On the one hand, (6.7) is nicely invariant over line 20, because during the execution of line 20 of c_i,

$$\forall j: s_{ij} = true \tag{6.8}$$

always holds, so that the setting of h_i to false in line 20 cannot invalidate any third term of (6.7) (this is where the fact that all h's are set to true in line 40j comes in). On the other hand, (6.7) fails to be invariant over line 30. The offending case is when line 30 is executed with h_i=false holding; (6.7) can then no longer be ensured.

But (6.6) and (6.7) are so "nearly enough" the solution of our problem that we should not let go of them lightly. We can attempt to find a suitable formula "between (6.6) and (6.7)" which is weak enough to be invariant and strong enough to imply (6.3). To this end we replace the h_k in (6.7) by a dummy term R_k:

$$\forall i: (eq_i \vee h_i \vee (\exists k: (R_k \,\&\, \underline{not}\ s_{ki}))) \qquad (6.9)$$

so that both (6.6) and (6.7) are special cases of (6.9) (with R_k=true and R_k=h_k, respectively).

Let us analyse in detail the obligation that (6.9) be weak enough to be an invariant. (6.9) is invariant over the first alternative in line 20 as long as (as with the h's in (6.7)) the setting of h_i to false cannot invalidate any third term of (6.9). The invariance of (6.9) over the second alternative, i.e. line 30, can be ensured if in addition the following holds:

$$\forall i: eq_i \vee R_i \qquad (6.10)$$

By postulating (6.10) can we thus avoid the failing cases of (6.7).

But (6.9) can itself be used to define the R's in such a way as to ensure (6.10)! (6.9) was to be our invariant, according to which $\underline{not}\ eq_i$ implies $h_i \vee (\exists k: (R_k \,\&\, \underline{not}\ s_{ki}))$. If we define the R_i such that the latter, in turn, implies R_i, then (6.10) is guaranteed, and with it the invariance of (6.9) over line 30. Let us formulate this condition on the R's:

$$\forall i: (h_i \vee (\exists k: (R_k \,\&\, \underline{not}\ s_{ki}))) \Rightarrow R_i \qquad (6.11)$$

This condition, to repeat, follows from our requirement that (6.9) be weak enough to be invariant.

The other requirement that (6.9) be strong enough to imply (6.3) can be translated simply into the requirement that the R's themselves be as strong as possible (i.e. as "false" as possible). We thus, in all, define the R's as the minimal solution of (6.11). The unique minimal solution can easily be constructed by setting the R_i to h_i in the first place and then switching the third term(s) of (6.11) to true whenever necessitated by a second term of (6.11). It remains to be shown that with the R's so defined, (6.9) is invariant over line 20. This is true because when h_i is set to false then R_i may become false but, because (6.8) holds, no other R_k becomes false.

This completes the proof because we have shown that (6.9) and (6.10) are invariants, and because if all h's are false then the minimal solution of (6.11) gives all R's false, which by (6.10) implies all eq's true as required. To see the connection to the proof given by Dijkstra in [33], we may rewrite (6.11) as a set of two formulae:

$$\forall i: \underline{\text{not}}\ h_i \vee R_i$$
$$\forall i,k: (\underline{\text{not}}\ R_k) \vee s_{ki} \vee R_i$$

and remark that these are the two formulae used by Dijkstra.

Because they satisfy (6.10) which is an analogue of (6.4), the R's behave "almost" as the h's do in program P6.2; they are, in all, a "slightly weaker" version of the h's. In fact the R's behave exactly as indicated in the program P6.4 below.

```
c_i:  do <∃j: h_j> →
        if <eq_i → h_i:=false; R_i:=∃k: R_k & not s_ki>
        [] <not eq_i → y_i:=f_i(y); ∀j: ((s_ij,R_j):=(false,true))>;
                                    ∀j: <h_j:=true; s_ij:=true>
        fi
      od
```

Program P6.4

This fact is another consequence of our arguments. In program P6.4 it is apparent in what way the R's are "slightly weaker" than the h's, and it is also clear that they satisfy (6.10).

6.2.3 A Control Sequence Proof

The purpose of this section is not to find any "better" proof of P6.1 (I don't think this is possible) but, first of all, to show that a rigorous proof in terms of control sequences can be given, and secondly, to enable a comparison between the two proofs (even though in this case the comparison is unfavourable for the control sequence proof).

I reproduce the program once more below.

```
10    do <∃j: h_j> →
20       if <eq_i → h_i:=false>
30       [] <not eq_i → y_i:=f_i(y)>;
40j                          ∀j: <h_j:=true>
         fi
      od
```

Program P6.1

We conduct the proof of (6.3) by contradiction. Suppose there exists a reachable state in which (6.3) does not hold, i.e. suppose there exists a valid execution

$$u = s_0 a_1 \ldots a_n s_n$$

of the form (5.10) such that both (5.11) and the validity property (5.14) are satisfied, and that in s_n the following holds true:

$$\forall j: \underline{\text{not}}\ h_j \quad \& \quad \exists i: \underline{\text{not}}\ eq_i \tag{6.12}$$

By assumption, all h's are true in s_0.

From these assumptions we construct a contradiction by working up backwards the sequence u. We use the letters l,p and q to denote indices in u and the letter i to denote indices in 1,...,N.

By assumption (6.12), there exists a component with index i0 such that $\underline{\text{not}}\ eq_{i0}$ and $\underline{\text{not}}\ h_{i0}$ in s_n.

Because $\underline{\text{not}}\ h_{i0}$ in s_n and because h_{i0} holds to start with, there must be an index l ($1 \leq l \leq n$) such that a_l is an execution of line 20 for c_{i0}. Define p0 as the index of the latest such a_l (i.e. p0 is maximal amongst these indices l).

Because $\underline{\text{not}}\ eq_{i0}$ in s_n there exists a component c_{i1} (possibly i1=i0) such that some a_{q0}, with $p0<q0$, is an execution of line 30 for c_{i1}.

We now consider the state s_{q0-1}, i.e. the state just before a_{q0}. We know that $\underline{\text{not}}\ eq_{i1}$ in this state because of the guard of a_{q0}. Suppose that h_{i1}=true in s_{q0-1}. In order to switch h_{i1} to false in s_n, all of the lines 40j for c_{i1} must occur after a_{q0} (otherwise u would not be a valid control sequence of c_{i1}!); in particular, h_{i0} is set to true by this, which contradicts the maximality of p0.

Therefore, in the state s_{q0-1} we have both $\underline{\text{not}}\ eq_{i1}$ and $\underline{\text{not}}\ h_{i1}$.

We can now repeat this argument to show the existence of indices p1 and q1 ($p1<q1<q0$) and a component c_{i2}, such that both $\underline{\text{not}}\ eq_{i2}$ and $\underline{\text{not}}\ h_{i2}$ hold in the state s_{q1-1} (the assumption h_{i2}=true in s_{q1-1} can be shown to contradict either the maximality of p0 or the

maximality of p1).

In sum, we thus construct a never-ending descending sequence of indices ...q2<q1<q0 in u, in contradiction to the finite length of u.

This completes the control sequence proof. Note that our convention to represent execution sequences as alternations between states and atomic actions has in this case proved convenient.

6.2.4 Discussion

The second proof using control sequences is quite evidently so much less elegant than the one using assertions that it might just as well immediately be forgotten. However I wish to use it to make a few general remarks.

The first point I wish to make is that our second proof, though "operational" in flavour, has not been based upon any "computational model" if the latter is understood to involve some kind of "interpreter" or "implementation" of our language. Both proofs have been completely rigorous and implementation-independent, the only difference being that they use different tools; also, both proofs have been phrased in terms of the program text only. Moreover, in the second proof we did not have to take into account an exploding number of execution sequences; rather, one general sequence was enough.

My second point is that though inelegant, the second proof may still be useful. For just as different proofs of the same mathematical theorem may expose different aspects about this theorem, so different proofs about a program may lead to different insights about the behaviour of this program. To illustrate this on our example: we might be interested in the fact that when, say, the second alternative of a component c_i is executed with not h_i holding (which is the "bad" case) then another component c_j ($j \neq i$) must previously have entered its second alternative with h_j holding. This fact follows immediately from our second proof but seems to be difficult to extract from the first proof.

On the other hand, the first proof exhibits clearly the connections between the two different programs P6.1 and P6.2, whereas no corresponding insights can be derived from the second proof.

The fact that assertional and operational proofs may both be acceptable rigorous and purely "textual" proofs of a program precludes any preference being placed on either method on the grounds of one of the methods being more "mathematical" than the other. In the example treated in the present section, the assertional method must clearly be favoured, while the (admittedly much more trivial) program in section 5.2.5 has been easier to prove in terms of execution sequences.

I suspect that in the long run the assertional method will prevail, simply because it allows one to argue explicitly in terms of "invariants", i.e. more or less "static" and "general" statements, which (given the exploding number and length of different executions) is what one, directly or indirectly, is forced to do anyway; an operational proof does of course not imply that one looks at all execution sequences one by one. I think that this example teaches us that we do not necessarily have to be afraid of using operational arguments, as these can be just as rigorous and effective as arguments phrased in terms of assertions. In the next section we study an example which, apparently most conveniently, uses a mixture of both types of arguments.

6.3 Finding an Euler Cycle

6.3.1 Introduction and Sequential Solution

Let an undirected, connected graph with N vertices and M edges be given. For every undirected edge e ($1 \leq e \leq M$) we introduce two opposite edges labelled e and -e, respectively. The problem we consider in this section is to find a single directed cycle which contains all directed edges, each edge exactly once; in other words, an Euler cycle. Because every vertex has the same number of incoming and outgoing edges, such cycles exist.

Ore in [87] describes the following sequential solution. Starting at an arbitrary vertex one follows a path, marking the traversed edges. When one arrives at some vertex for the first time the entering edge is marked especially. When one reaches a vertex one always follows next an edge which has not previously been marked. However, the opposite of the entering edge should be followed only as a last resort when there are no other edges available.

This simple algorithm has been known for a very long time. The famous "Ariadne's thread" method of finding one's way out of a labyrinth is based on a rather similar principle. It is however not particularly general, in the sense that, while some Euler cycle is indeed a result, there are Euler cycles which could never be a result of its application. A more general sequential algorithm is described in [62] in connection with the problem of enumerating and classifying all possible Euler cycles.

The sequential algorithm described above, traversing as it does all edges sequentially, is of time efficiency proportional to the number M. Since for connected graphs, $N-1 \leq M \leq N*(N-1)/2$, the algorithm thus needs maximally time proportional to N^2. When learning of this algorithm, I thought it nice and simple enough to be transformed into an "equivalent" (or otherwise related) concurrent algorithm, both as an exercise in atomic actions and with the aim of improving its efficiency. This section describes the results of these efforts.

A reasonable way to decompose the problem, it seems, is by assuming that at each vertex of the graph a process resides and that these processes communicate in some way via the edges of the graph. A solution built in this way can indeed be found. This solution is described in section 6.3.2. The same solution is also described, proved and analysed in [10], where it is shown to be of time efficiency O(N). We do not repeat its proof in section 6.3.2; instead we concentrate on its design, showing how a set of "local" arguments can guide us in the discovery of a solution which has the desired "global" effect.

When the arguments leading to this solution are scrutinised further, it turns out that there are other possible strategies than

locating the processes in the vertices of the graph. We show this in section 6.3.3 where, starting with our first concurrent solution, we derive a second solution in which the processes are located at the edges, rather than the nodes, of the graph. This gives more processes but also a greater degree of concurrency.

In section 6.3.3 we shall prove the correctness of the second algorithm in detail. The proof uses a mixture of invariant-assertion and control-sequence type arguments. It may therefore be interesting as an example for the convenient use of the two proof methods in combination. Finally, in section 6.3.4 we show that our algorithms are still not "general" in the afore-mentioned sense that every Euler cycle can be a result.

6.3.2 Concurrent Solution Using Vertex Processes

Let the graph be given in some form, say by its incidence matrix. For every directed edge $e \in \{-M,\ldots,-1\} \cup \{1,\ldots,M\}$, let head(e) and tail(e) denote its head and tail vertices, respectively. Let further, for any vertex V, the set of its incoming and outgoing edges be denoted by IN(V) and OUT(V), respectively. We wish to define a bijection "suc" on the set of edges such that by starting with an arbitrary edge and following "suc" one obtains an Euler cycle. In particular, "suc" must satisfy the equation

$$head(e) = tail(suc(e))$$

for all edges e.

As we have said earlier, we decide (a priori) to locate our processes at the vertices of the graph; thus we have to implement a concurrent program consisting of N sequential components. We derive design heuristics for this program by comparing it to the sequential algorithm. Consider a vertex V. In the sequential algorithm, when V is first reached, say through the edge $e \in IN(V)$, then the Euler cycle to be constructed will lead through all outgoing edges of V and only then back

through the "last exit" -e. If V is left via an edge e' ∈ OUT(V) then the same applies for the vertex head(e').

It is therefore plausible to let the processes associated with V be responsible for the following task. Let a special "entrance" edge e to V be given. The process in question considers it to be its responsibility to connect e to all so far "untraversed" outgoing edges of V (in some order), finally leading back to -e. If all vertex programs behave that way, an Euler cycle will ultimately result. To mark the special entrance edges we may use a colouring; say, all edges are "black" initially and become "white" when they could serve as an entrance to their head vertices.

The question is then: when should edges become white? The whole process should start somewhere. We therefore arbitrarily select an initial vertex and add to it a single incoming edge, call it e_0, (and its opposite $-e_0$) which is initialised to being "white". In general, an edge should turn white when it could serve as an entrance for its head vertex, i.e. when its tail vertex has "traversed" it. We must have exclusion if the head and tail vertices of a pair of untraversed edges which are both black, both want to "traverse" these edges simultaneously.

It remains to be decided what should happen in case two (or more) incoming edges of V are marked as "white" simultaneously. This means that the tail vertices of these edges expect that their opposites are returned to. It apparently suffices that V chooses an arbitrary white incoming edge and makes sure that all other white incoming edges are connected back to their opposites. This is the rationale behind our first solution (see program P6.5 below).

Before giving the solution we have to decide the precise form in which the graph is given initially. This is rather arbitrary, and in [10] I have decided to let the incidence matrix be contained in the initial value of "suc". This can be done by initialising "suc" in such a way that by selecting an incoming edge of a vertex V and alternatively applying the functions suc,-("minus"),suc, etc., one can enumerate all adjacent edges of V.

As an example, we consider the following graph and the initial configuration in which the function suc is represented by dotted lines:

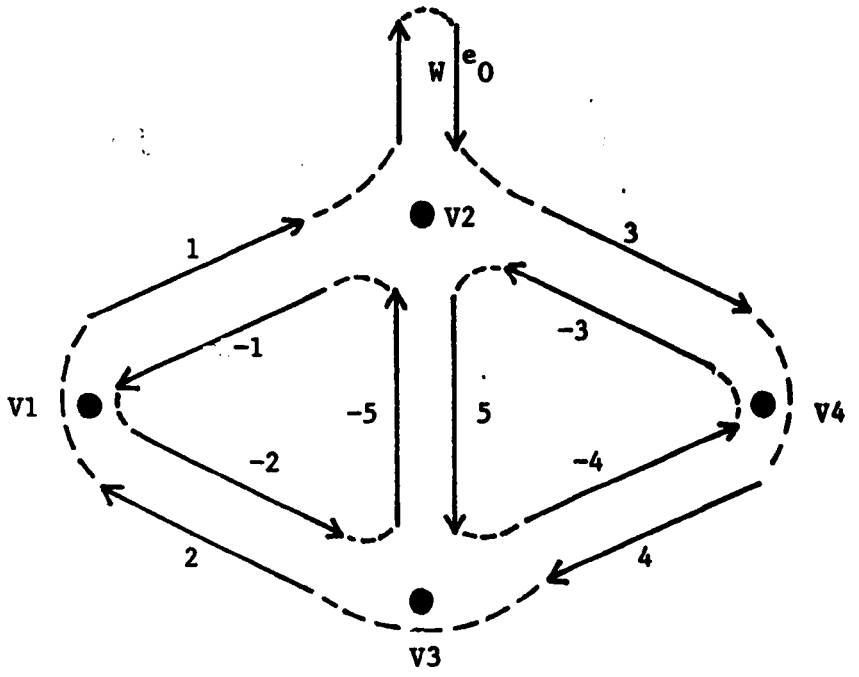


Figure 6.1: An Initial Configuration

Initially, only the edge e_0 is white. We denote the colour of an edge e by col(e), and "B" denotes "black" while "W" denotes "white".

Under these assumptions, the algorithm for the vertex V becomes:

```
local var e,j1,j2: {-M,...,-1}∪{1,...,M};

if ∃ e ∈ IN(V): col(e)=W →
   "choose an arbitrary e ∈ IN(V) with col(e)=W";
   j1:=e; j2:=suc(j1);
   do j2≠-e → if <col(-j2)=B → col(j2):=W>;
                                j1:=-j2; j2:=suc(j1)
               [] col(-j2)=W  → suc(j1):=suc(-j2);
                                suc(-j2):=j2;
                                j2:=suc(j1)
               fi
   od
fi
```

Program P6.5

Program P6.5 is formally proved in [10] where it is also shown that it produces an Euler cycle with O(N) average time efficiency.

Note that in P6.5 we have made extensive use of the local variable rule of section 5.2.4. Even though the suc function looks like a global variable, every component accesses only the suc of incoming edges, so that suc can be partitioned into disjoint portions local to the vertex programs.

The program finishes with all edges white if and only if their opposite edges are black, and an Euler cycle contained in suc. For example, under the assuption that the vertex programs are executed in the following order:

V2; V3; V1; V4

and the choices in the third line of P6.5 are settled so that

$e[V2]=e_0$, e[V3]=5, e[V1]=-1, e[V4]=3,

then the following Euler cycle results from an application of P6.5 to the initial configuration shown in Figure 6.1.

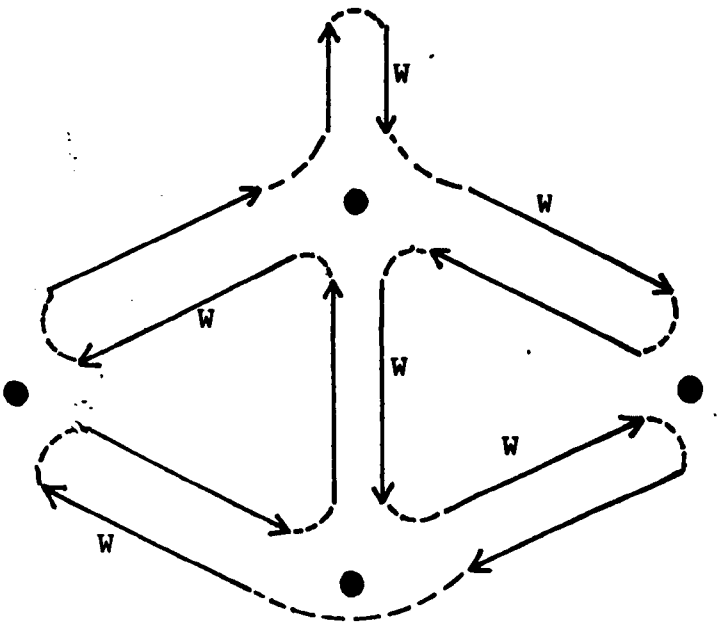


Figure 6.2: A Final Configuration

6.3.3 Concurrent Solution Using Edge Processes

In our previous solution, the order in which the outgoing edges of a vertex are traversed was determined by the initial configuration. We could equally well have left this order to be decided by a non-deterministic command within the program. A less obvious change in the program can be motivated by the following reasoning.

Consider a vertex V with a number of white incoming edges (Figure 6.3(a)). In P6.5, one of these white incoming edges is chosen as the "entrance" e to V to which all untraversed edges of V (indicated by dotted lines in Figure 6.3) are to be connected, while the other white incoming edges of V are immediately turned back (Figure 6.3(b)).

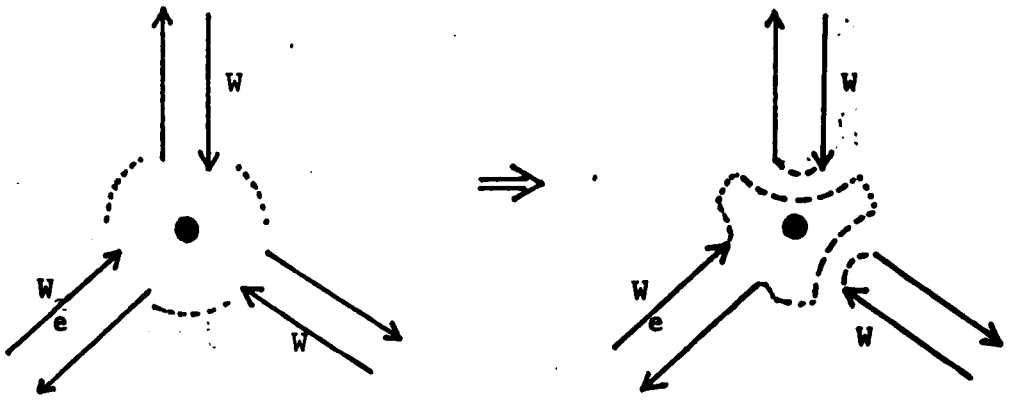


Figure 6.3

Rather than connecting all of the untraversed edges just to e, we could also allow some of them to be connected to some of the other white incoming edges. This would leave the salient "external" properties of the vertex program undisturbed but would lead away from the rather arbitrary preferential treatment of one particular white incoming edge.

Let us think about implementing this strategy. A vertex process could at any time maintain as many pairs of indices {j1,j2} (as in P6.5) as there are white incoming edges. This could be implemented by splitting a vertex process into several parallel subprocesses, one for each white incoming edge, each maintaining a (set of) local pointer(s) and trying to connect (in competition with the others) as many untraversed edges as possible to "its" edge. From this reasoning, only one further step leads to the realisation that we do not need the vertex processes at all as separate entities but rather, that we can think directly in terms of the new edge processes. Because every edge could become a white incoming edge of its head vertex, we need a separate process for each edge of the graph, to be activated only if that edge turns white.

The fine details of such a process can now readily be implemented. Because the edge e must eventually lead back to -e, it is reasonable to start with suc(e)=-e and gradually to extend the path from e to -e with further untraversed edges of V=head(e). Thus we assume that initially

$$\forall e:\ suc(e)=-e \qquad (6.13)$$

and that e_0 is again the only edge which is white (all others are black). The incidence matrix of the graph is given by the sets IN and OUT. For the purposes of "linking in new edges" our edge processes maintain local variables "j". Our new solution, associating a process with every edge e, is shown as program P6.6 below.

```
local var j: {-M,...,-1}∪{1,...,M} (initially j=-e);

10   if <col(e)=W> ->
20      do <UNTR(head(e))≠∅ ->
30         "choose e´ ∈ UNTR(head(e))";
40         (suc(e),j,suc(-e´)) := (e´,e´,j);
50         col(e´) := W>
        od
     fi,
```

where UNTR(V) = {e´ ∈ OUT(V) | col(e´)=col(-e´)=B}

Program P6.6

"UNTR(V)" stands for the "set of untraversed edges of V".

We prove P6.6 in two steps, (6.14) and (6.15):

There is a suc-cycle which contains precisely all white edges and their opposites; (6.14)

Eventually, an edge is white iff its opposite is black. (6.15)

This suffices because under the assumption (6.15), the cycle which exists by (6.14) is an Euler cycle. We prove that (6.14) is invariant and then, using control sequences, we prove (6.15).

(6.14) is true initially because there is, by (6.13), a cycle containing just the only white edge e_0 and its opposite $-e_0$. But (6.14)

also remains true over the atomic action in lines 20-50 of P6.6 because the conditions under which the new white edge is created in line 50 ensure its truth; the new edge and its opposite are just "merged" into the cycle that already exists.

It remains to prove (6.15). We shall show that a state in which there exists a pair of untraversed edges (i.e. edges which are opposite and both black) cannot be a final state. Because the graph is connected, if such edges exist at all then there exists, in particular, a pair of untraversed edges bordering on some white edge.

Suppose col(e´)=col(-e´)=B and suppose e is a white incoming edge, say, of head(e´) (the argument for tail(e´)=head(-e´) is, of course, symmetric). Then the execution producing this state is not "complete" in the sense of section 5.2.2 because its projection onto the edge process of e is not a complete control sequence of the latter (if it were then we would have UNTR(head(e))=UNTR(head(e´))=∅, in contradiction to both e´ and -e´ being black). On the other hand, suppose that e is a white <u>outgoing</u> edge of head(e´). Then we can use the following auxiliary invariant:

$$\forall V: (\exists e' \in OUT(V): col(e')=W) \Rightarrow (\exists e \in IN(V): col(e)=W) \qquad (6.16)$$

in order to deduce that head(e´) must also have a white incoming edge, and the argument can be repeated. The invariance of (6.16) follows from its initial truth and the fact that a new white outgoing edge can only be created under the condition (line 10!) that there already exists a white incoming edge.

6.3.4 Discussion

The proof of P6.6 could quite nicely be partitioned into an invariant (6.14), stating that an assertion is "always" true, and a termination condition (6.15), stating that "eventually" some other assertion becomes true. We have proved the former in the assertional style while the latter was proved by showing that it is a consequence of the

"completeness" condition for execution sequences. I have found that this follows a pattern of proof which can frequently be encountered, and that the termination condition frequently involves a more explicitly "operational" argumentation than the invariant.

I would like the proof of P6.6 to be seen as further evidence that operational arguments, whenever convenient, may without loss of rigour be used. Our proof of (6.15) can certainly not be accused of being "too informal". It requires but a "mechanical" transformation for the reasoning contained in it to be couched in terms of the control sequence formalism; this can be done to any desired degree of formality.

The three programs described in sections 6.3.1-6.3.3 are progressively more general in the sense that any application of the first and second program, leading to certain Euler cycles, can be simulated by applications of the second and third program, respectively, leading to the same cycles. The two concurrent algorithms are more general than the sequential one, since they can produce the Euler cycle shown in Figure 6.2 which cannot result from an application of the sequential algorithm.

However, the last algorithm is still not completely general because it cannot produce the Euler cycle shown in Figure 6.3 below.

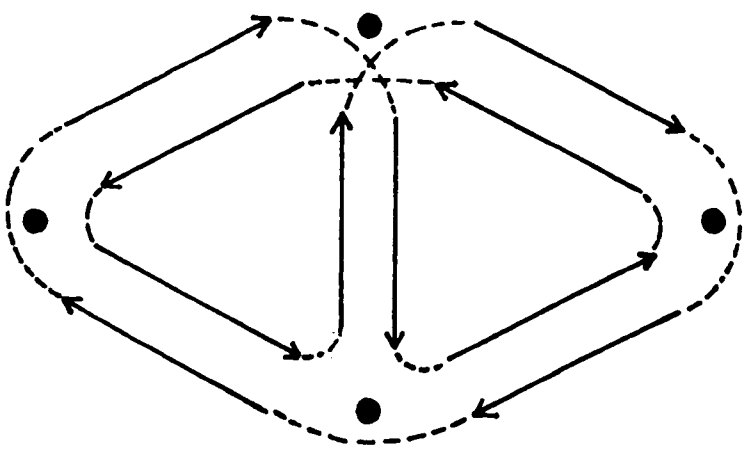


Figure 6.3

I have so far not succeeded in finding a completely general concurrent algorithm. However, I think this should be possible, perhaps by starting with Kasteleyn's algorithm [62] rather than with the one given in section 6.3.1.

Another possible generalisation concerns the graph itself. Any directed graph in which the vertices have the same number of incoming and outgoing edges possesses Euler cycles; our present algorithms however work only for a subclass of such graphs. I think a generalisation along these lines should not prove too difficult.

6.4 A Fail-Safe Distributed Extrema-Finding Algorithm

6.4.1 Introduction

In a small but very nice paper [25], Chang and Roberts have presented a simple distributed algorithm for finding the highest numbered node in a circular distributed system. Suppose that N nodes ($N \geq 1$) are arranged in a ring such that each node can send messages only to its immediate successor and (by implication) receive messages only from its immediate predecessor. Thus a node may send messages to itself only by having them pass through all of the other nodes. We refer to the direction of message passing as "clockwise". The nodes are assumed to be numbered from 1 to N so that no two nodes have the same number; however, the distribution of node numbers on the ring is arbitrary (in particular, does not have to be clockwise ascending). The problem is for a node i, having no knowledge about the other nodes, to determine whether or not it has the highest number, i.e. whether or not $i=N$.

In the present section we reconsider this problem and give a detailed proof for the solution by Chang and Roberts in section 6.4.2. Furthermore, we extend their algorithm to make it work also in the case in which one or more nodes fail. We consider only very "kind" failures by which the circular nature of the network is not destroyed (section 6.4.3).

This section is included for a number of reasons. Firstly, it shows an application of the invariant assertion method for a distributed algorithm. Secondly, we discuss how a "properly terminating" program containing partial deadlock could be converted into one which "terminates properly" in the sense of section 5.2.2, i.e. with all component processes terminating. Thirdly we have here an algorithm which can most conveniently be expressed in a CSP[52]-like notation but which can also easily be translated into a "shared data" program as defined in section 5.2. We discuss this transformation in particular with a view to generalisation.

Let us define our problem more precisely. We assume N nodes to be given which are linked via one-frame input/output buffers. Every node has precisely one input buffer from which it can read and one output buffer into which it can write. Of course, the output buffer of a node is the input buffer of the immediately succeeding node. The buffers being "one-frame" means that their state can be described by a Boolean variable "empty".

Every node has at its disposal two actions denoted by !E and ?V, where E is an expression and V a variable. !E can be executed if the output buffer is empty and effects the transfer of the value of E to the buffer and a switch of its state to empty=false. ?V can be executed if the input buffer is not empty and effects the transfer of the value contained in this buffer to V and a switch to empty=true. This understanding makes the input/output actions !E and ?V somewhat different from corresponding actions in [52] where output and matching input are required to occur coincidentally ("handshake synchronisation"); more about this can be found in the next section.

6.4.2 The Basic Algorithm

Under the assumptions given in section 6.4.1, the Chang/Roberts algorithm can be written down as follows (program P6.7), where we abbreviate "do true → c od" to "do c od".

```
Node i:        local var x: integer;

          10   !i;
          20   do ?x;
          30      if x<i -> skip
          40      [] x=i -> "success"
          50      [] x>i -> !x
                  fi
               od
```

Program P6.7

P6.7 works as follows. Initially, all node numbers start circulating (line 10). They keep circulating (lines 20 and 50) until they are taken out of circulation by being dropped at nodes with a higher number (line 30). Only the highest number completes a full circle and its eventual return to the sender indicates success (line 40). The program terminates with node N executing "success" and all other programs waiting for more input.

Before proceeding to prove the correctness of P6.7 formally, I wish to make two remarks. Firstly, even though the deadlocking of one component has in section 5.2.2 been taken as implying the non-termination of the entire program, I wish to point out that we can in the present case without problem speak of the "proper termination" of the entire program, even though eventually all but one components end up in deadlock. If the reader feels uncomfortable about this then he can easily transform the present program into one in which all components terminate. To this end we may introduce a special value called "found" for x, which is not an integer, and transform P6.7 into P6.8 below:

```
        local var x: integer ∪ {"found"};
                  exit, its_me: Boolean;

5       (exit,its_me) := (false,false);
10      !i;
20      do not exit -> ?x;
23                     if x="found" -> !x;
27                                     exit:=true
30                     [] x≠"found" & x<i -> skip
40                     [] x≠"found" & x=i -> !"found";
43                                           exit:=true;
47                                           its_me:=true
50                     [] x≠"found" & x>i -> !x
                       fi
        od
```

Program P6.8

Program P6.8 terminates with exit=true for all components and its_me=true only for node N.

The construction applied here, namely a special message sent out by a terminating process to notify all others of its termination, can evidently be generalised. The only precondition for its general applicability seems to be that the network in question must be strongly connected, so that the special message can be disseminated to every other node. We do not explicitly transform the remaining programs in this section according to this construction but assume that the reader can do this for himself if he so desires.

My second remark is that the program P6.7 can easily be transformed into one which uses global variables and atomic actions. To this end we define a global array "buffer" (or "b" for short) as follows:

```
var b: array 1..N of record value: integer,
                            empty: Boolean
             end.
```

(Initially, b[j].empty=true for all 1≤j≤N.)

For every node i we denote its input buffer by b[in,i] (or just b[in] if i is known from the context) and its output buffer by b[out,i] or just b[out]. The two actions !E and ?V can be replaced by atomic actions combined with an **await** command utilising our **await** interpretation. The new program becomes, for node i:

```
        local var x: integer;

10      <b[out].value:=i;
         b[out].empty:=false>;
20      do if <not b[in].empty → x:=b[in].value;
                                 b[in].empty:=true>
           fi;
30         if x<i → skip
40         [] x=i → "success"
50         [] x>i → if <b[out].empty → b[out].value:=x;
                                       b[out].empty:=false>
                    fi
           fi
        od
```

Program P6.9

The programs P6.7 and P6.9, under our given assumptions, are evidently equivalent. A priori, I would prefer P6.9 (even though it is longer) because I know its precise semantics from section 5.2.2. This reason disappears as soon as we have defined the commands !E and ?V just so that P6.7 is indeed equivalent to P6.9. We could also have defined !E and ?V differently, for example as acting on a K-frame buffer (K>1); the translation into atomic actions would then not have presented any problems either.

Suppose that !E and ?V are defined on a "0-frame buffer", i.e.

output and matching input are coincident and none can occur without the other occurring at the same time. This "handshake synchronisation" is just the semantics given to !E and ?V in [52], and I would now like to consider the question whether such operations can be translated into atomic actions. Unfortunately this does not seem to be quite as straightforward as in the K-buffer case for $K \geq 1$. However I think the following transformation will do the trick. Let c_1 and c_2 be two processes which are to communicate via the follwing two commands.

Figure 6.4

Thus, c_1 outputs, in a "synchronised assignment", the expression E into the variable x of c_2.

We introduce two shared Boolean variables p_1 and p_2 (=false initially) which are to be used only in connection with the communication involving the two matching commands shown in Figure 6.4. We rewrite these two commands, and introduce a third intermediate process c_{12} (which contains the "synchronised assignment" explicitly), as shown in Figure 6.5.

```
c1:   .                          c2:   .
      .                                .
      .                                .
  <p1:=true>;                      <p2:=true>;
  if <not p1 -> skip> fi           if <not p2 -> skip> fi
      .                                .
      .                                .
      .                                .

c12:  if <p1 & p2 -> x:=E;
                     (p1,p2):=(false,false)>
      fi
```

Figure 6.5

If necessary, c_{12} could be placed in a loop. This construction seems to be rather generally applicable, leading to the transformation of every CSP-type program into a program using shared variables and atomic actions as defined in section 5.2.

This conclusion actually seems to contradict a result by de Nicola et al [85], to the effect that there exist CSP-type systems which cannot be expressed in terms of shared data. However, their result holds good for a rather specialised abstract model, and it is not clear (to me, at least) that programs written in CSP or, for that matter, our notation can readily be described by their models. This question, in general, needs more researching. At any rate, the reader will find no difficulty in translating all programs in this section into programs using shared data and atomic actions. This being understood, we now conduct our discussions interchangeably using either the !E and ?V notation or explicitly the (one-frame) buffer variables b[.].

After these deviations, we come back to our main concern, which is the formal proof of P6.7. We reproduce P6.7 below for convenience.

```
10    !i;
20    do ?x;
30       if x<i -> skip
40       [] x=i -> "success"
50       [] x>i -> !x
         fi
      od
```

Program P6.7

We wish to establish the truth of the following two correctness criteria:

If node i succeeds then i=N; (6.17)

Eventually, a node succeeds. (6.18)

Alternatively, in terms of the variables contained in program P6.7, (6.17) and (6.18) could be replaced by the requirement (or "specification", in the terms of section 2.5) that eventually

i=N <=> its_me[i]=true,

where its_me[i] denotes the local Boolean its_me of component i.

We shall establish (6.17) and (6.18) by means of a series of invariants, and we shall find it helpful to prove the termination condition (6.18) by the introduction of some auxiliary variables. We first need a way of talking about a node number, j say, being "on the ring". We say that j is on the ring iff either there exists a buffer containing j (i.e. the buffer is not empty and its value equals j) or there exists a node, i say, whose variable x[i] equals j. We immediately have the following invariant:

Every node number j, $1 \leq j \leq N$, is at most once on the ring. (6.19)

Proof: (6.19) holds after line 10 which ensures that every node number

is exactly once on the ring. The sequence of input/outputs in lines 20/50 (in case x=j) cannot disturb the truth of (6.19) because it simply effects the propagation of j on the ring.

For any two nodes y and z, we define j to be "(clockwise) between y and z" if between y and z there is a buffer containing j, or between y (exclusive) and z (inclusive) there is a node i such that x[i]=j. For example, supposing j is exactly once on the ring then j must be either between y and z or between z and y.

Our next invariant expresses the intuitive notion that no node number can proceed further than its (clockwise) nearest higher successor. Let for $y \neq N$, nexthigh(y) denote the nearest higher node; let nexthigh(N)=N. Then

$\forall$ j: the number j is between
the node j and the node nexthigh(j). (6.20)

Proof: line 10 clearly establishes (6.20) to begin with, and (6.20) cannot be violated by line 50 because of its guard. As a special case of (6.20), we deduce that the number N is always on the ring. (6.17) also follows from (6.20). Suppose node i succeeds and $i \neq N$. Then nexthigh(i)$\neq$i and (6.20) is violated for j=i.

The intuitive argument for (6.18) runs as follows. Because every execution of a loop cycle furthers the circulation of some node number on the ring, and because (as we shall see) there can be no global deadlock, some node must eventually see its own number again. More precisely, a given node number j can be propagated exactly as far as to the node nexthigh(j) defined above. To capture this behaviour, it is convenient to define, for each j, an integer function t(j) expressing the "distance" between the number j and the node nexthigh(j). We initialise t(j) to

t(j) = number of nodes between the node j (exclusive)
and the node nexthigh(j) (inclusive)

and amend the program so that t(j) is decreased whenever the number j is being propagated (see P6.10 below). We further claim that t(j) stays non-negative for all j. This can be deduced from the assertions in P6.10.

```
10    !i;
20    do ?x; {t(x)>0}
25       t(x):=t(x)-1; {t(x)≥0 & (t(x)=0 => x≤i)}
30       if x<i → skip
40       [] x=i → "success"
50       [] x>i → {t(x)>0}
                  !x
         fi
      od
```

Program P6.10

Using t(j) we can define the total termination function

$$t = \sum_{1}^{N} t(j)$$

which is properly decreased each time a node executes its loop, and $t \geq 0$ always on account of $t(j) \geq 0$ for all j.

All that is left to be proved is that there can be no deadlock. On the ring, a global deadlock means that either

(a) all buffers are empty and all nodes are waiting for input, or

(b) all buffers are full and all nodes are waiting for output.

(a) cannot occur because (as a consequence of (6.20)) N is always on the ring, and (b) cannot occur because, while all buffers are full initially, before any more numbers can be put on the ring, at least as many numbers have had to be taken away.

6.4.3 A Fail-Safe Algorithm

We consider only what Chang in [24] calls "clean failures". That is to say, we demand that the failure of a node leaves the circular nature of the network intact; that no messages can be destroyed by a failure; and that failed nodes do not recover. Such a failure can be modelled by a node entering (and remaining in) a "failure mode" in which it just performs a shift

do ?x;!x od

from its input buffer to its output buffer. To avoid the possible loss of messages, we assume that a failure mode cannot be entered while an input is being processed in the normal way.

Our problem is to revise the basic algorithm so that it works correctly for clean failures. More precisely, let LIVE denote the set of all non-failed nodes (LIVE={1,...,N} initially) then our fail-safe solution should satisfy the following two correctness criteria:

If node i succeeds then i=max(LIVE); (6.21)

Eventually, either LIVE=$\emptyset$ or a node succeeds. (6.22)

We first note that (6.21) and (6.22) are generalisations of (6.17) and (6.18), respectively. We also observe that unless the number max(LIVE) fails to be on the ring the basic algorithm already correctly implements (6.21) and (6.22).

My intention was to use this close correspondence between these respective correctness criteria in order to derive our fail-safe solution in a more or less systematic way. This was in fact the main reason why I chose to prove the comparatively trivial statements (6.17) and (6.18) to such an excessive degree of detail in section 6.4.2. Unfortunately, owing to lack of time I cannot devote much thought on such a derivation. I therefore just present the solution and sketch its proof.

```
        local var x,m: integer;

 5      m:=i;
10      !i;
20      do ?x;
30         if x<i -> skip
40         [] x=i -> "success"
50         [] x>i -> if x<m -> skip
53                   [] x=m -> !i
57                   [] x>m -> m:=x
                     fi;
60                   !x
           fi
        od
```

Program P6.11

Algorithm P6.11 works by singling out the highest number N as a "restart" mechanism. By the time a node i enters line 53, we have m[i]=N and we can furthermore deduce that node N has failed and that the number i is no longer on the ring. Hence node i can propose itself again by re-emitting its own number.

The detailed correctness argument runs as follows. The proof of (6.21) is but a trivial extension of the argument for (6.17) contained in section 6.4.2, with N appropriately replaced by max(LIVE). For (6.22) we need a slightly generalised termination function which may become infinite iff LIVE=$\emptyset$ (the system then enters an infinite loop).

The absence of deadlock can also be shown. The ring can never become empty because the number N remains on it. Overcrowding could only occur as a consequence of the two successive outputs in lines 53 and 60; however, every time line 53 is entered, it is known that the previously emitted number "i" has been taken off the ring, so that there is at least one "gap" in the ring which ensures further progress. This ends the proof sketch for P6.11. It also ends our chapter describing a few examples.

7. CONCLUSION AND GENERAL DISCUSSION

What has been achieved and what remains to be done? In conclusion, I would like to address this question. Also, I wish to discuss why I think that the work described in this thesis can be relevant, and in particular I wish to discuss some of the wider issues which I consider to form part of the context of this work.

I consider that we have achieved the development of a comprehensive formal framework for the semantic understanding of the programs under consideration. In particular, we have arrived (I claim) at an adequate formal understanding of atomic actions.

We are in possession of the syntax and the semantics of a small language supporting atomic actions, and we know roughly how this language can be extended (say, with more complex data construction facilities, but also with more elaborate kinds of atomic actions). In giving its semantics, we have been led by the idea that there is a connection between atomic actions and relational semantics, and we have in detail explored this connection. We also know what it means for such programs to be (partially or totally) correct with respect to a given input/output specification.

Furthermore, we have given the programmer not only a formal semantics which he can resort to when in doubt about the behaviour of his program, but we have also shown on several examples how concurrent programs could be designed and how their correctness could be proved. For some examples we have given alternative proofs and we have explored the differences between these proofs. I have taken pains to make the correctness arguments as clear as possible to myself; of course this does not imply that I claim that programs "ought to" be designed or proved that way.

We have also provided the implementor with a criterion as to how programs using atomic actions can be implemented; namely, the executions they generate should satisfy our dynamic atomicity criteria. We have shown that, and why, this does not necessarily mean that atomic actions are mutually excluded, thus giving the implementor a chance to find more

concurrent implementations (we have however not considered concurrency in detail).

Finally, we have also shown connections to other work. In particular, we have discussed the relation to the Owicki/Gries proof method. We have made the point that our semantics involves "control" in a direct way, rather than indirectly as the Owicki/Gries method, but that this does not mean that it is based on any particular "model of computation". Also, our semantics gives rise to an alternative but nonetheless precise proof method.

We have also briefly compared concurrent programs using atomic actions with concurrent programs using other mechanisms for interaction, such as CSP programs. I have substantiated my belief that "shared data" programs are so general that CSP programs can be "simulated" by them; which is not to say a lot, because even sequential non-deterministic programs are so general as to "almost" be able to simulate shared data programs. The "almost" refers to the "finite delay property", i.e. the fact that one can write concurrent programs which under the fairness assumption (which we have adopted) cannot be simulated by boundedly non-deterministic programs.

With all of this, my own curiosity about atomic actions is, in the main, satisfied. I have given my answer to the question of what it means that an action "occurs as if instantaneously"; the answer is that this action behaves, as far as its environment is concerned, in accord with its effect relation, as the latter can be calculated independently by looking only at the action concerned. As the answer, in short, I would submit formula (5.27) in section 5.3. p 161

There is perhaps one further important point which I am still uncertain about in connection with atomic actions. This concerns the question of the semantic independence of atomic actions, and the related question of information flow between actions, which have been discussed, but not solved, in section 5.4. In my opinion, variable transformations as defined in section 5.4 could become a further important tool in the design of programs, comparable perhaps to the use of the predicate calculus. There is no reason why we should always have to work with a

constant set of variables which have been fixed at the beginning of the program.

Throughout this thesis, I have tried to provide, at least for my own benefit, a set of clearly defined "interface" points at which questions which have been answered give rise to new questions which have to be investigated. Some of these questions are related to the theme of the thesis and are considered by me to be important and interesting. I wish to recollect these questions.

In section 2.6 we have defined the AND operator as a formal dual to the non-deterministic OR operator. We have maintained that AND (which is also related to the "backtrack choice") can conveniently be understood as a special "parallel" operator effecting the "splitting up" of the entire state space into disjoint copies. We have advanced the formula (2.28) for the relational semantics of programs containing both AND and OR.

Because AND is a "parallel" operator the question arises as to how it relates to the parallel operator || which we have used throughout the rest of the thesis. On the one hand, it would seem to be possible to interpret AND as a special case of ||, the former expressing "disjoint" (non-interfering) parallelism, and the latter expressing possibly interacting parallelism. This could perhaps be done by interpreting every AND program as a program using || whose initial state space has been duplicated as many times as there are final state spaces for the AND program.

On the other hand, it would also appear that programs using || and OR are special cases of programs using AND and OR. Intuitively, this is so because the semantics (5.5) of shared data programs is, via (2.29) and (2.28´), a subset of (2.28). But in this "argument" we have not taken into account that (5.5) is not rich enough to express the abortion of individual components, as opposed to the abortion of the whole program. (2.28), however, can take account of this: abortion of a single branch in an AND program just means that this branch does not contribute to the end result. I am fairly certain that the connection between AND and || can be explored further, possibly to the point of yielding a

unified (relational) framework for concurrent, non-deterministic and backtrack programs (which sounds nice).

A second problem area I am keen on knowing more about concerns infinite computations in general. We have now a collection of properties all of which relate to infinite computations: continuity (in general), bounded non-determinacy (i.e. continuity of the wp), the finite delay property in various forms (in particular, the maximality property of section 5.2.2 and Park's "fair merge" property [90]), several properties of infinite occurrence nets such as K-density (others are discussed in [14] and in [103]), and, last not least, computability. We know of connections between some of these properties, but there is a feeling that more, and more important, connections can be made; I am for instance uneasily aware that my argumentation in section 3.4 is not very "stringent", in the sense that it does not capture very precisely the intuitive point that the ban on processes which "make a choice out of infinitely many possibilities" should not be extended to concurrent processes.

Another issue, already mentioned above, which I consider to be important is the semantic independence between actions. Perhaps related to this, I would like to see in detail how data construction facilities could best be incorporated into our small language. I have long thought that one could possibly find a richer language allowing the, in some sense, "harmonic" definition of variables together with atomic actions which operate on these variables, perhaps à la "abstract data types". Again, this has to be left to future work.

In my opinion the most serious issue that has to come under the heading "to be investigated" is the actual design of concurrent programs. With the immense complexity of even "small" concurrent programs in mind, it seems to me absolutely essential that there be not only formal semantics, correctness criteria and proof methods which the programmer can apply a posteriori in order to verify his program, but that there also be a set of a priori heuristics, or guidelines, which may aid the programmer in developing a correct program. Such guidelines may well be considered even more important than the necessary formal

semantic framework by itself. I am therefore rather unhappy that I have not been able to work out such a set of guidelines.

Design heuristics cannot be derived by abstract reasoning in terms of sets and functions. Rather, we need to have a lot of programming experience, both in writing small programs and in studying the interactions in larger programs. Once a certain amount of experience has been accumulated, it may then perhaps be possible, by abstraction, to extract a few principles which may guide the design of other programs.

Owing to extensive experience with sequential programming, it is by now possible to discern a few rather general principles which are helpful in programming. These are, by and large, common knowledge. I might mention the "linear search theorem" [30] which implies that if we want to find the least element in an array with a certain property then we best search through the indices in ascending order. Another extremely useful principle is that of "weakening predicates", which can frequently be employed to design a loop. This principle states that it is often possible to find a weaker version of the (unary or binary) predicate describing a given specification, which can serve as a loop invariant. In [17], the development of a simple program using this principle is described.

As far as concurrent programming is concerned, all I can say is that I expect these principles to be just as helpful as they have turned out to be for sequential programming. Indeed both in section 6.2 and in section 6.4 we were led to "weaken" certain predicates in order to obtain more general programs; in one case (the transition from program P6.2 to program P6.1) the generalisation consisted of an increase in concurrency, in the other case (the transition from program P6.7 to program P6.11) it consisted of added fault tolerance.

Because my experience with concurrent programming is limited to a few small examples, I find it hard to abstract and think of principles other than the above, which could serve as design heuristics. However I can clearly discern the need for these. Namely, there are design decisions which need to be made in concurrent programming but which do not arise in sequential programming. For example, the question may arise

how many sequential components the target concurrent program is going to consist of (indeed, whether it is at all appropriate to split the solution into sequential components), and how these components can be distributed over the "problem area". This question came up in section 6.3 where our initial decision to associate processes with the vertices of the graph could not a priori be justified.

So much for my answer to the question raised at the beginning of this chapter. I had planned the remainder of this chapter to contain the following points (a)-(f). Unfortunately, I was not able to finish a detailed text with which I could be satisfied. I therefore only describe very briefly the line of argument I would have wished to take.

(a) I agree with Dijkstra's verdict in [31] that the program specifications form a "logical firewall" between purely mathematical statements on the one hand ("does the program meet its specification?") and less formally defined questions on the other hand ("do the specifications adequately reflect whatever use they are going to be put to?").

(b) I disagree with De Millo, Lipton and Perlis [83] in their conclusion that "program verification is futile". One of the mistaken assumptions these authors make in their article is that program specifications are of neccessity informal.

(c) The argument for verification stands and falls with the assumption that there be formal specifications and formal semantics for our programs. I consider that the work described in this thesis shows that formal specifications and formal semantics can be defined for concurrent programs just as well as for sequential programs, thus forestalling arguments that concurrent programs are somehow "inherently unverifiable"; they are only usually much harder to verify than sequential programs of comparable length.

(d) Dijkstra maintains in [31] that the relationship between programs and their specifications is "amenable to scientific treatment" while the questions associated with the other side of the wall, i.e. the adequacy of specifications, are not. I disagree with this use of the term

"scientific"; rather, I would say that the relationship between programs and specifications is a mathematical one while those other questions involve non-mathematical considerations. "Science", to my mind, covers much more than just mathematics. The question whether a specification fulfils its purpose adequately is not per se non-scientific.

(e) Dijkstra (while explicitly avoiding to judge the relative "importance" of the two issues) goes on to consider it a waste of time for the scientist to expend his thoughts on what he calls "the non-scientific issue". I am afraid I cannot go along with this. What good are all our nice formal methods, our techniques for effective reasoning and our beautiful proof systems, if the use they are eventually put to is not in itself good?

(f) With Weizenbaum [101], I am of the opinion that the scientist should consider himself at least partly responsible for the products created by other people using the methods he has developed. I think he should not close his eyes either to the good use or to the misuse which his beautiful theories may be undergoing. No one should delude himself that a theory, merely by virtue of being beautiful, is invulnerable against misuse. The most beautifully correct program could evoke an explosion under which more than one proponent of program verification would die. This is why I consider it not only time well spent but also of utmost importance that we should start caring about the end results of our efforts: to create the conditions under which beautiful theories can safely be developed.

A. Appendix

A.1 Usual Symbols

1. Set Theoretic Symbols

X,Y are sets.

$\in$	element of
$\notin$	not element of
$\subseteq$	subset of
$\subset$	proper subset of
$X \times Y$	Cartesian product
$X \cap Y$	intersection
$X \cup Y$	union
$X \setminus Y$	difference
$\|X\|$ or card(X)	cardinality of X
$\emptyset$	empty set
{...}	set brackets
$\bigcap \{...\}$	intersection quantifier
$\bigcup \{...\}$	union quantifier

2. Logical Symbols

&	logical "and"
$\vee$	logical "or"
$\neg$ or not	negation
=>	implication
<=>	equivalence
=	equality
$\stackrel{=}{}$	equality by definition
$\forall$	for all... (universal quantifier)

$\exists$ there exists... (existential quantifier)

3. Functional Algebra

X, X_1, X_2, Y are sets.

$f: X \longrightarrow Y$ f is a function from X to Y

Let $g: X_1 \times X_2 \longrightarrow Y$ be a function from $X_1 \times X_2$ to Y

Then by $g(x_1,.): X_2 \longrightarrow Y$ (for fixed $x_1 \in X_1$)

and $g(.,x_2): X_1 \longrightarrow Y$ (for fixed $x_2 \in X_2$)

we denote the two projection functions derived from g

4. Relational Algebra

S a set, R, R_1, $R_2 \subseteq S \times S$ relations over S, $x,y \in S$.

$(x,y) \in R \iff x\ R\ y$

R^* transitive reflexive closure of R

R^+ transitive non-reflexive closure of R

$Dom(R) = \{x \in S \mid \exists y: x\ R\ y\}$ domain of R

$Cod(R) = \{y \in S \mid \exists x: x\ R\ y\}$ codomain (range) of R

$xR = \{y \in S \mid x\ R\ y\}$ image of x under R

$Ry = \{x \in S \mid x\ R\ y\}$ coimage of y under R

$R_1 \cap R_2$ intersection

$R_1 \cup R_2$ union

$R_1 \circ R_2$ relational composition
$(x,y) \in R_1 \circ R_2 \iff \exists z \in S: (x,z) \in R_1 \ \&\ (z,y) \in R_2$

$\emptyset$ empty relation

id identity relation

al = S x S all relation

head(x,y) = x

tail(x,y) = y

A.2 Letter Index

A	atomic activity
B	set of conditions in an occurrence graph or occurrence net (chapters 3 and 4) condition in an *if* or *do* clause (chapters 2 and 5)
E	set of events in an occurrence graph or occurrence net
F	set of arrows in a net
G	goal of a non-deterministic program binary predicate
H	goal of a backtrack program (section 2.6)
I	author
L	set of levels in a structured occurrence graph
M,$\mathfrak{M}$	markings
N	constant number
$\mathbb{N}_0$	set of nonnegative integers
P,Q	unary predicates
S	state space set of places in a net (only section 3.2)
T	set of transitions in a net (only section 3.2) set of activities in a structured occurrence graph
X,Y	subsets of the state space
X	set of "elements" in a net (only chapter 3)
a	atomic action (chapter 5)
b	conditions
c	a program ("command") a cut (section 3.4)
e,e´,e´´	events
f	a function
m(c)	relational meaning of c
m,n	numbers

r	a read event
s,t	states
s'	initial state
s	a place (section 3.2)
t	a transition (section 3.2)
u	an execution sequence (section 5.2)
w	a write event
x,y,z	variables in a program

References

1. T. Anderson and P.A. Lee, Fault Tolerance: Principles and Practice, Prentice Hall (1981).

2. G.R. Andrews, "Parallel Programs: Proofs, Principles and Practice," CACM Vol. 24(3), pp.140-146 (March 1981).

3. K.R. Apt and E.R. Olderog, "Proof Rules Dealing With Fairness," Bericht No. 8104, Christians-Albrecht-Universitaet Kiel (March 1981).

4. R.J. Back, "Semantics of Unbounded Nondeterminism," Lecture Notes in Computer Science Vol. 85, pp.51-63, Springer Verlag (1980).

5. J. de Bakker, "Semantics and Termination of Non-Deterministic Recursive Programs," pp. 435-477 in Automata, Languages and Programming, ed. S. Michaelson and R. Milner, Edinburgh (July 1976).

6. J. de Bakker, Mathematical Theory of Program Correctness, Prentice Hall Series in Computer Science (1980).

7. J. Barwise (ed.), Handbook of Mathematical Logic, Springer Verlag (1977).

8. C. Berge, Graphs and Hypergraphs, North Holland, Amsterdam (1973).

9. E. Best, "A Note on The Proof of a Concurrent Program," IPL Vol. 9(3) (October 1979).

10. E. Best, "Proof of a Concurrent Program Finding Euler Paths," Lecture Notes in Computer Science Vol. 88, pp.142-153, Springer Verlag (September 1980).

11. E. Best, "Atomicity of Activities," Lecture Notes in Computer Science Vol. 84, pp.225-250, Springer Verlag (1980).

12. E. Best, "Notes on Predicate Transformers and Concurrent Programs," TR 145, Computing Laboratory, University of Newcastle upon Tyne (1980).

13. E. Best, "A Theorem on the Characteristics of Non-Sequential Processes," Fundamenta Informaticae Vol. III(1), pp.77-93 (July 1980).

14. E. Best, "The Relative Strength of K-Density," Lecture Notes in Computer Science Vol. 84, Hamburg, pp.261-276, Springer Verlag (1980).

15. E. Best, "Adequacy of Path Programs," Lecture Notes in Computer Science Vol. 84, Hamburg, pp.291-305, Springer Verlag (1980).

16. E. Best and B. Randell, "A Formal Model of Atomicity in Asynchronous Systems," Acta Informatica (to appear) (1981).

17. E. Best and F. Cristian, "Systematic Detection of Exception Occurrences," TR 165, Computing Laboratory, University of Newcastle upon Tyne (1981).

18. G. Birkhoff, Lattice Theory, 1948.

19. J.R. Bitner and E.M. Reingold, "Backtrack Programming Techniques," CACM Vol. 18(11), pp.651-656 (November 1975).

20. H.J. Boom, "A Weaker Precondition For Loops," TR 104/78, Mathematisch Centrum, Amsterdam (1978).

21. M. Broy, P. Pepper, and M. Wirsing, "On Relations Between Programs," Lecture Notes in Computer Science Vol. 83, Paris, pp.59-78, Springer Verlag (1980).

22. M. Broy, "Are Fairness Assumptions Fair?," Proceedings of 2nd International Conference on Distributed Computing Systems, Paris, pp.116-125 (April 1981).

23. A.K. Chandra, "Computable Non-Deterministic Functions," Proceedings of the 19th IEEE Symposium on Foundations of Computer Science, Ann Arbor, MI, pp.127-131 (1978).

24. E. Chang, "Decentralised Algorithms in Distributed Systems," TR-CSRG-103, Computer Systems Research Group, University of Toronto (October 1979).

25. E. Chang and R. Roberts, "An Improved Algorithm for Decentralised Extrema-Finding in Circular Configurations of Processes," CACM Vol. 22(5), pp.281-283 (May 1979).

26. E. Cohen, "Information Transmission in Sequential Programs," Proceedings of Workshop on Foundations of Secure computation, Atlanta (1977).

27. J. Cohen, "Non-Deterministic Algorithms," Computing Surveys Vol. 11(2), pp.79-94 (June 1979).

28. F. Cristian, "Robust Data Types," Acta Informatica (to appear) (1981).

29. E.W. Dijkstra, "Cooperating Sequential Processes," pp. 43-112 in Programming Languages, ed. F. Genuys, Academic Press, New York (1968).

30. E.W. Dijkstra, A Discipline of Programming, Prentice Hall (1976).

31. E.W. Dijkstra, "A Position Paper on Software Reliability," ACM SIGSOFT, Software Engineering Notes, pp.3-5 (October 1977).

32. E.W. Dijkstra, "On a Political Pamphlet From the Middle Ages," ACM SIGSOFT, Sofware Engineering Notes Vol. 3(2), pp.14-16 (April 1978).

33. E.W. Dijkstra, "Finding the Correctness Proof of a Concurrent Program," Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen Vol. 81(2), pp.207-215 (June 1978).

34. E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens, "On-the-Fly Garbage Collection: An Exercise in Cooperation," CACM Vol. 21(11), pp.966-975 (November 1978).

35. E.W. Dijkstra, "My Hopes of Computer Science," Proceedings of Fourth Software Engineering Conference, Munich, pp.442-448 (1979).

36. G.W. Ernst and E.F. Ogden, "Specification of Abstract Data Types in MODULA," ACM TOPLAS Vol. 2(4), pp.522-543 (October 1980).

37. R. Eswaran, J. Gray, R. Lorie, and I. Traiger, "On the Notions of Consistency and Predicate Locks," CACM Vol. 19(11), pp.624-633 (November 1976).

38. R.W. Floyd, "Assigning Meanings to Programs," Applied Mathematics Vol. 19, AMS Providence (1967).

39. R.W. Floyd, "Non-Deterministic Algorithms," JACM Vol. 14(4), pp.636-644 (October 1967).

40. F. Furtek, "The Logic of Systems," PhD Thesis, MIT (1976).

41. H.J. Genrich, K. Lautenbach, and P.S. Thjagarajan, "Elements of General Net Theory," Lecture Notes in Computer Science Vol. 84, Hamburg, pp.21-163, Springer Verlag (1980).

42. H.J. Genrich and E. Stankiewicz-Wiechno, "A Dictionary of Some Basic Notions of Net Theory," Lecture Notes in Computer Science Vol. 84, pp.519-535, Springer Verlag (1980).

43. S.L. Gerhart and L. Yelowitz, "Control Structure Abstractions of the Backtracking Programming Technique," IEEE Transactions on Software Engineering Vol. SE-2(4), pp.285-292 (December 1976).

44. J.N. Gray, "Notes on Data Base Operating Systems," pp. 394-481 in Operating Systems, Springer Lecture Notes in Computer Science, Springer Verlag (1978).

45. D. Gries and G. Levin, "Assignment and Procedure Call Proof Rules," ACM TOPLAS Vol. 2(4), pp.564-579 (October 1980).

46. P. Guerreiro, "A Relational Model for Non-Deterministic Programs and Predicate Transformers," Lecture Notes in Computer Science Vol. 83, Paris, pp.136-146, Springer Verlag (1980).

47. A.N. Habermann, Introduction to Operating System Design, SRA Computer Science Series (1976).

48. M.H.T. Hack, "Decidability Questions for Petri Nets," TR-161, MAC, MIT, Boston (June 1976).

49. D. Harel, "First-Order Dynamic Logic," Lecture Notes in Computer Science Vol. 68, Springer Verlag (1979).

50. M. Hennessy and E.A. Ashcroft, "The Semantics of Nondeterminism," pp. 478-493 in Automata, Languages and Programming, ed. S. Michaelson and R. Milner, Edinburgh (July 1976).

51. C.A.R Hoare, "Towards a Theory of Parallel Processing," pp. 61-72 in Operating Systems Techniques, Academic Press, New York (1972).

52. C.A.R Hoare, "Communicating Sequential Processes," CACM Vol. 21(8), pp.666-677 (August 1978).

53. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," CACM Vol. 12, pp.576-583 (1969).

54. C.A.R. Hoare and P.E. Lauer, "Consistent and Complementary Formal Theories of the Semantics of Programming Languages," Acta Informatica Vol. 3, pp.135-153 (1974).

55. C.A.R. Hoare, "Some Properties of Predicate Transformers," JACM Vol. 25(3), pp.461-480 (July 1978).

56. A. Holt, "A Petri-Net Based Theory of Choice," Technical Report ADS 617/245-9540, Applied Data Research, Massachussetts (1977).

57. J.J. Horning and B. Randell, "Process Structuring," Computing Surveys Vol. 5(1), pp.5-30 (March 1973).

58. R. Janicki, "A Characterisation of Concurrency-Like Relations," Lecture Notes in Computer Science Vol. 70, Evians-les-Bains, pp.108-122, Springer Verlag (1979).

59. R. Janicki, "An Algebraic Structure of Petri Nets," Lecture Notes in Computer Science Vol. 83, pp.177-192, Springer Verlag (1980).

60. P. Johansen, "Non-Deterministic Programming," BIT Vol. 7, pp.298-304 (1967).

61. R.M. Karp and R.E. Miller, "Parallel Program Schemata," JCSS Vol. 3, pp.147-195 (1969).

62. P.W. Kasteleyn, "Graph Theory and Crystal Physics," in Graph Theory and Theoretical Physics, ed. F. Harary, Academic Press, London (1967).

63. S. Katz and Z. Manna, "Logical Analysis of Programs," CACM Vol. 19(4), pp.188-206 (April 1976).

64. R.M. Keller, "Formal Verification of Parallel Programs," CACM Vol. 19(7), pp.371-387 (July 1976).

65. J.R. Kennaway and C.A.R. Hoare, "A Theory of Nondeterminism," Lecture Notes in Computer Science Vol. 85, pp.338-350, Springer Verlag (1980).

66. J. King, "Program Correctness: On Inductive Assertion Methods," IEEE Transactions on Software Engineering Vol. SE-6(5), pp.465-479 (September 1980).

67. L. Lamport, "The `Hoare Logic' of Concurrent Programs," Acta Informatica Vol. 14, pp.21-37 (1980).

68. B.W. Lampson and H. Sturgis, "Crash Recovery in a Distributed Data Storage System," Xerox PARC Internal Memorandum (1979).

69. A. van Lamsweerde and M. Sintzoff, "Formal Derivation of Strongly Correct Concurrent Programs," Acta Informatica Vol. 12, pp.1-31 (1979).

70. P.E. Lauer, "Consistent Formal Theories of the Semantics of Programming Languages," PhD Thesis, TR 25.121, IBM Laboratory, Vienna (November 1971).

71. P.E. Lauer and R.H. Campbell, "Formal Semantics for a Class of High-Level Primitives for Coordinating Concurrent Processes," Acta Informatica Vol. 5, pp.247-332 (1975).

72. P.E. Lauer, E. Best, and M.W. Shields, "On the Problem of Achieving Adequacy of Concurrent Systems," in Formal Description of Programming Concepts, ed. E. Neuhold, North Holland (1978).

73. P.E. Lauer, P.R. Torrigiani, and M.W. Shields, "COSY - a System Specification Language Based on Paths and Processes," Acta Informatica Vol. 12, pp.109-158 (1979).

74. K. Lautenbach, "Dual Aspects of Process Coordination," Interner Bericht 74/04, GMD-ISF (1974).

75. K. Lautenbach and E. Pless, "Grundmuster der Koordination von Systemen," pp. 251-279 in Graphen, Algorithmen, Datenstrukturen, Applied Computer Science, Hanser Verlag, Munich (1976).

76. C. Lengaur, "Title Not Known At Time of Writing," PhD Thesis, University of Toronto, Canada (1981).

77. D.B. Lomet, "Process Structuring, Synchronisation and Recovery Using Atomic Actions," Proceedings of the ACM Conference on Language Design for Reliable Software ACM SIGPLAN Notices Vol. 12(3), pp.128-137 (March 1977).

78. Z. Manna, "The Correctness of Programs," JCSS Vol. 3, pp.119-127 (1969).

79. Z. Manna, Mathematical Theory of Computation, McGraw Hill Computer Science Series (1974).

80. G. Memmi and G. Roucairol, "Linear Algebra in Net Theory," Lecture Notes in Computer Science Vol. 84, Hamburg, pp.213-223, Springer Verlag (1980).

81. P.M. Merlin and B. Randell, "Consistent State Restoration in Distributed Systems," TR 133, Computing Laboratory, University of Newcastle upon Tyne (October 1977).

82. P.M. Merlin and B. Randell, "Consistent State Restoration in Distributed Systems," Digest of Papers FTCS-8, Toulouse, pp.129-134 (June 1978).

83. R.A. De Millo, R.J. Lipton, and A.J. Perlis, "Social Processes and Proofs of Theorems and Programs," CACM Vol. 22(5), pp.271-280 (May 1979).

84. R. Milner, "A Calculus of Communicating Systems," Lecture Notes in Computer Science Vol. 92, Springer Verlag (1980).

85. R. de Nicola, A. Martelli, and U. Montanari, "Communication Through Message Passing Or Shared Memory; A Formal Comparison," Second International Conference on Distributed Computing Systems, Paris, pp.513-522 (April 1981).

86. M. Nielson, G. Plotkin, and G. Winskel, "Petri Nets, Event Structures and Domains," Lecture Notes in Computer Science Vol. 70, Springer Verlag (1979).

87. O. Ore, Theory of Graphs, American Mathematical Society, Colloquium Publications, Rhode Island (1962).

88. S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I," Acta Informatica Vol. 6, pp.319-340 (1976).

89. S. Owicki and L. Lamport, "Proving Liveness Properties of Concurrent Programs," Op. 57, Stanford University/SRI (October 1980).

90. D. Park, "On the Semantics of Fair Parallelism," Lecture Notes in Computer Science Vol. 86, Copenhagen, pp.504-524, Springer Verlag (1980).

91. J.L. Peterson, "Petri Nets," Computing Surveys Vol. 9(3), pp.223-252 (September 1977).

92. C.A. Petri, "Kommunikation mit Automaten," PhD Thesis, Schriften des IIM No 2, Rheinisch-Westfaelische Universitaet, Bonn (1962). Also: English Translation, Grifiss Air Force Base, RADC-TR-65-377, Vol.1 (1966)

93. C.A. Petri, "Grundsaetzliches zur Beschreibung diskreter Prozesse," 3. Colloquium ueber Automatentheorie, Basel, Birkhauser Verlag (1967).

94. C.A. Petri, "General Net Theory and Communication Disciplines," Computing Systems Design: Proc. Joint IBM/Univ. of Newcastle upon Tyne Seminar (1976).

95. C.A. Petri, "Non-Sequential Processes," Technical Report ISF-77-05, GMD St. Augustin, W. Germany (1977).

96. C.A. Petri, "Concurrency," Lecture Notes in Computer Science Vol. 84, Hamburg, pp.251-260, Springer Verlag (1980).

97. B. Randell, P.A. Lee, and P.C. Treleaven, "Reliability Issues in Computing System Design," Computing Surveys Vol. 10(2), pp.123-165 (June 1978).

98. M.W. Shields, "Adequate Path Expressions," Lecture Notes in Computer Science Vol. 70, Springer Verlag (1979).

99. A. Silberschatz and Z. Kedem, "Consistency in a Hierarchical Database System," JACM Vol. 27(1), pp.72-80 (January 1980).

100. M. Wand, "A Characterisation of Weakest Preconditions," JCSS Vol. 15, pp.209-212 (1977).

101. J. Weizenbaum, Computer Power and Human Reason, W.H. Freeman Co. (1976).

102. G. Winskel, "An Exercise in Processes With Infinite Pasts," Proceedings of the First European Workshop on Theory and Applications of Petri Nets, Strasbourg (September 1980).

103. G. Winskel, "Events in Computation," PhD Thesis, University of Edinburgh (August 1980).

104. N. Wirth, Algorithms + Data Structures = Programs, Prentice Hall (1976).

105. A.W. Holt et al., "Information System Theory Project," Final Report, RADC-TR-68-305, NTIS AD 676972, Princeton, N.J. (1968).

106. J.B. Rothnie et al., "Introduction to SDD-1," ACM Transactions on Database Systems Vol. 5(1), pp.1-17 (March 1980).