Abstract:

We propose a formal syntax and a formal relational semantics for concurrent programs. Our notation is an extension of guarded commands with the parallel operator and the atomic action feature. Our semantics define the meaning of a program by means of a relation between input states and output states, and is based on a precise notion of an execution sequence. We discuss relationships to other work, in particular to sequential non-deterministic (wp) semantics and to the Owicki-Gries proof method. Finally we show how our semantic model can be applied in the proofs of some substantial example programs, both as an alternative to, and in combination with, assertional proofs.

* This paper has also been published as:


† Author's present address:

Gesellschaft für Mathematik und Datenverarbeitung mbH Bonn,
ISF, Postfach 1240, Schloss Birlinghoven, D-5205 St. Augustin 1.

Relational Semantics of Concurrent Programs
(with some Applications)

By

E. Best†

TECHNICAL REPORT SERIES

Series Editor: Mr. M.J. Elphick

No. 180
July, 1982

© 1982 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.
BEST, Eike

Relational semantics of concurrent programs (with some applications). [By] Eike Best.

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1982.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 180).

Added entries
UNIVERSITY OF NEWCASTLE UPON TYNE.

Suggested classmarks (primary classmark underlined)
Dewey (18th):

001.6424

U.D.C.:

519.682

Suggested keywords
ATOMIC ACTIONS
CONCURRENCY
CONCURRENT PROGRAMS
FORMAL SEMANTICS
PROGRAM VERIFICATION

Abstract

We propose a formal syntax and a formal relational semantics for concurrent programs. Our notation is an extension of guarded commands with the parallel operator and the atomic action feature. Our semantics defines the meaning of a program by means of a relation between input states and output states, and is based on a precise notion of an execution sequence. We discuss relationships to other work, in particular to sequential non-deterministic (wp) semantics and show how our semantic model can be applied in the proofs of some substantial example programs, both as an alternative to, and in combination with, assertional proofs.

About the author

Dr. E. Best was a Research Associate in the Computing Laboratory from 1976 to 1981, and received his Ph.D. in February 1982. He is now a member of the research staff at GMD in Bonn.
In section 3, illustrating various aspects of the model, we concentrate on the proof that the wp semantics is, except for our fairness assumption, a special case of our semantics. The proof occupies section 3.6 and Appendices A and B. Further, we outline some syntactic extensions (section 3.1); we argue that in the presence of concurrency the if command should be interpreted as an await (section 3.2); we give criteria for the correctness of concurrent programs (section 3.3); we argue that atomic actions are characterised by their effect relations (section 3.4); we discuss our fairness postulate (section 3.5); we sketch the relationship to the Owicki-Gries proof method (section 3.7); finally, we explain our choice of the definition of an execution sequence (section 3.8).

Section 4 contains four example programs in whose proofs our method will be applied. Three of these examples are rather non-standard. We deal with a fixpoint program by Dijkstra [DIJ78] in section 4.2, comparing his assertional proof with a proof conducted using our semantics. In section 4.3 we develop a parallel Warshall algorithm while section 4.4 describes a parallel algorithm finding Euler paths. Both of these algorithms will be proved correct using a combination of assertional and operational methods.

Finally, section 5 contains a short discussion and an outlook to future work. The mathematical notation used in this paper can be found in Appendix C.

2. SYNTAX AND SEMANTICS OF A CONCURRENT LANGUAGE

We envisage our concurrent programs to consist of parallel combinations of a number of sequential components. These sequential components may involve the usual (non-deterministic) sequential control operators, i.e. the catenation operator "\|", the alternative clause "if-fi", and the repetitive clause "do-od". Moreover, they may be structured into atomic actions within which unrestricted nesting is allowed.

2.1 Syntax

We use the following abbreviations for syntactic met vowels: PROG=concurrent program (start symbol), SEQPROG=sequential program, ELPROG=elementary program, IF=alternative clause, DO=repetitive clause, GC=guarded command, GCLIST=list of GC, V=variable, E=expression, B=Boolean expression, TAIL=sequence of atomic actions.

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

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

(SYN3) ELPROG ::= <abort> / <skip> / <V:=E> / <PROG> / IF / DO

(SYN4) IF ::= if GCLIST fi

(SYN5) DO ::= do GC od

(SYN6) GCLIST ::= GC / GC\|GCLIST

(SYN7) GC ::= <B>\|TAIL / <B>\|PROG / <B>\|PROG;TAIL

(SYN8) TAIL ::= <PROG> / <PROG>\|TAIL.

For V, E and B no syntax will be given.

Any syntactic entity enclosed within the brackets <> will be called an "atomic action". Atomic actions will be given separate names, regardless of whether or not they "look" the same. For example, the two atomic actions in P1 below will receive two different names.

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

Program P1

In general, concurrent programs c derivable from PROG have the following form:

\[ c = c_1 \| ... \| c_n \]  \hspace{1cm} (2.1)

where the \( c_i \) (1\leq i \leq n) are derivable from SEQPROG. We call the \( c_i \) the "sequential components" of \( c \).
For an atomic action \( a \) in \( c \) we denote by \( \text{encl}(a) \) its immediately enclosing atomic action (or \( c \), if \( a \) is an outermost action). Note that \( \text{encl}(a) \) is well-defined since our syntax requires atomic actions to be well-nested. The set of outermost actions will be denoted by \( A = \{ a | \text{encl}(a) = c \} \). For \( a \in A \), we denote the component \( c_i \) in which \( a \) is contained by \( \text{cpt}(a) \). Again, \( \text{cpt}(a) \) is well-defined because our syntax and naming convention ensure that every \( a \in A \) is contained in exactly one component of (2.1). By convention, all variables are global integer variables unless otherwise specified. As usual, the state space of \( c \), denoted by \( S \), is defined as the set of mappings from variables of \( c \) to values. \( S \) will be assumed countable.

Our rules for enclosing parts of \( c \) in atomic action brackets are unusually restrictive. Compare P2 (see below) which is syntactically allowed, with P3 which is not.

\[
\text{Program P2}
\]

\[
\text{skip} \quad \text{||} \quad \langle y := x + 1 \rangle \quad \text{||} \quad \langle z := x \rangle
\]

\[
\text{Program P3}
\]

\[
\text{skip} \quad \text{||} \quad y := x + 1 \quad \text{||} \quad z := x
\]

These restrictive rules are useful for the definition of our semantics. Having defined the semantics, any redundant brackets (for example, as in P2) can then safely be omitted. We will make some use of this possibility in our examples.

2.2 Semantics

The semantics of \( c \) will be defined as a relation

\[
m(c) \leq S \times (S \cup \{ \perp \})
\]

(2.2)

where \( \perp \) is the usual non-termination symbol. For the sake of simplicity we assume the evaluation of all expressions \( E \) and \( B \) to terminate and to lead into the domain of \( V \) if used in the context of an assignment \( V := E \).

(2.2) will be defined by induction, giving the definition of \( m(c) \) directly for basic programs and guards \( B \), whereafter \( m(a) \) can be assumed known for \( a \in A \). Thus, induction will be over atomic actions in \( A \), rather than over the sequential components \( c_i \). This approach is in accord with the intuitive idea (see \( [\text{BES80}, \text{BES81a}] \)) that an atomic action is characterised by its effect relation. See section 3.4 for a discussion of this.

Let \( s' \in S \) be an arbitrary initial state and let \( s \in S \cup \{ \perp \} \). We define:

\[
s' m(\text{abort}) = \{ \perp \}
\]

(2.3)

\[
s' m(\text{skip}) = \{ s' \}
\]

(2.4)

\[
s' m(V := E) = \{ s \}, \text{ where } s(V) = \text{value of } E \text{ in } s', \text{ and } s(W) = s'(W) \text{ for all variables } W \neq V.
\]

(2.5)

\[
s' m(B) = \begin{cases} \{ s' \} & \text{if } B(s') = \text{true} \\ \emptyset & \text{if } B(s') = \text{false} \end{cases}
\]

(2.6)

\[
(s', s) \in m(B \cdot c) \iff B(s') \land (s', s) \in m(c).
\]

(2.7)

For a program \( c \) of the general form (2.1) the definition of \( m(c) \) will be given in two parts. In (2.12) below we define the conditions for a proper final state \( s \in S \) to be reachable from \( s' \), while in (2.17) we define the conditions for \( \perp \) to be reachable.

First, let \( c_i \) be one of the sequential components of \( c \). We define the set of "control sequences" (abbreviated c.s.) and "complete control sequences" (c.c.s.) associated with \( c_i \) as strings of atomic actions. The intuitive meaning is that a c.s. represents a partial "trace" (in the sense of [Hoa78a]) of \( c_i \) while a c.c.s. represents a complete "trace"; for example, a loop may have an infinite partial trace, but all complete traces of a loop will be defined finite and containing the negation of the guard as their last element.

It is understood throughout that the empty string is a c.s. (but not a c.c.s.) of any program.
Finite maximal non-complete executions characterise deadlocks. Infinite non-maximal executions characterise a set of non-fair executions, as will be examined more closely in section 3.5. Using this notion of maximality we define for \( s' \in S \):

\[
(s', 1) \in \text{em}(c) \quad \Rightarrow
\exists u = s_0a_1 \ldots a_T S_T: s' = s_0, \ s_T = 1, \text{ and } u \text{ is an execution,}
\]

or \( \exists \) maximal non-complete execution starting with \( s' \).

The second requirement in (2.17) encompasses both deadlock (for a finite maximal non-complete execution) and fair non-termination (for an infinite maximal execution). The fairness property enshrined in (2.17) means, for example, that \( P6 \) (see below) is required to terminate.

\[
\langle x := 1 \rangle \\
\mid \mid \text{ do } \langle x := 0 \rangle \langle x := y + 1 \rangle \text{ od}
\]

Abbreviating "\((x,y) = (p,q)\)" to "\((p,q)\)", the infinite sequence

\[
(0,0) \langle x := 0 \rangle \langle 0,0 \rangle \langle y := y + 1 \rangle \langle 0,1 \rangle \langle x := 0 \rangle \langle 0,1 \rangle \langle y := y + 1 \rangle \ldots
\]

satisfies both (2.10) and (2.13) but fails to be maximal in the sense of (2.16): \( \langle x := 1 \rangle \) is infinitely often enabled but does not occur. \( P6 \) has been discussed, amongst others, by Hoare in [HOA78b] and by Park in [PAR80]. Our definition accords with the intuition expressed in [PAR80].

3. GENERAL REMARKS AND CONSISTENCY CONSIDERATIONS

3.1 Syntactic Extensions

Our syntax restricts the way in which atomic actions can be placed within programs. In particular, every opening bracket must be matched uniquely by a closing bracket. However, one may write perfectly sensible programs which violate this restriction. One can also write sensible programs in which atomic actions and other control structures, such as alternative clauses and loops, overlap. Consider the following examples.

\[
\langle 12x := y; \\
\quad \text{if } x = 0 \text{ alt } 1 \rangle \\
\quad \text{if } x = 0 \text{ alt } 2 \\
\text{fi}
\]

Program P7

\[
\langle 12z := 1; \\
\quad \text{do } z + N \text{ alt } 12 \to <\text{body}>; \\
\quad <2z := z + 1 \text{ alt } 2 \\
\text{od}
\]

Program P8

In P7, the opening bracket \( \langle 12 \) has two closing brackets, \( >1 \) and \( >2 \). In P8, the closing bracket \( >12 \) belongs to two different opening brackets, \( <1 \) and \( <2 \).

Both P7 and P8 may make semantic sense. It is easy to incorporate such programs in our formalism. Despite their awkward syntactic form, the set of (complete) control sequences can be defined as before. For P7 we have two possible c.c.s., namely \( \langle x := y ; x = 0 \text{ alt } 1 \rangle \) and \( \langle x := y ; x + 0 \rangle \langle \text{alt } 2 \rangle \). For P8, we have \( \langle z := 1 ; z + N \rangle \) or \( \langle z := 1 ; z + N \rangle \), followed by zero or more instances of \( \langle \text{body} > z := z + 1 ; z + N \rangle \), followed by \( \langle z := z + 1 ; z + N \rangle \).

Such programs can therefore be handled in exactly the same way as described in section 2. The general principle is to allow as "sequential components" all programs whose executions give rise to sequences of atomic actions, i.e., for which c.s. and c.c.s. are defined. This includes P7 and P8 but excludes (in line with the intuition expressed in [RAN78]) the proper overlapping of atomic actions. Analogous remarks hold true for other syntactic extensions, for instance if one wishes to allow (as does, for example, [LAM80]) assignments and/or expressions to be broken up into smaller atomic actions.
3.2 The "Wait" Command

Our semantics is such that the clause if B... fi has the effect of a "wait" if B is false. This is because control sequences other than ones which lead to the execution of an enabled guard have simply not been defined. Two alternative definitions would also have been possible: the "abort option" (as in guarded commands) whereby disabled clauses may lead to abortion, or the "skip option" (as in Algol 60) whereby disabled clauses may be equivalent to "skip".

The reason for adopting the "wait" interpretation, rather than any of the other two options, is threefold. Firstly, the "wait" semantics, in terms of control sequences, turns out to be simpler than those of the other two options: both for the abort option and for the skip option the definition of additional control sequences would be necessary. Secondly, the "wait" is of practical significance. For example, the semaphore operation P(x) is now simply equivalent to if <x>0;x:=x-1> fi. Thirdly, both the abort option and the skip option can easily be programmed explicitly by adding, respectively, the clauses "not B-abort" or "not B-skip". The same is not true vice versa: under any of the other interpretations, a "wait" can only be programmed by an explicit wait loop.

3.3 Correctness Criteria

The correctness of a program is a notion which is well-defined only with respect to a specification. We first define, in the style of [BES81b], a specification G (for "goal") as a relation G ∈ S × S with the following interpretation. For an initial state s' ∈ S within the domain of G we require the intended implementation of G to terminate in a final state s satisfying G (i.e., (s', s) ∈ G), and for initial states outside Dom(G) we don't care what c does. (What may happen when the latter requirement is strengthened is described in [BES81b]).

Thus we define a program c "correct with respect to G" iff

\[ ∀s' ∈ \text{Dom}(G): s'm(c) ≤ s'G. \]  (3.1)

This corresponds to the notion of "total correctness" as defined, for example, in [MAN74]. The notion of "partial correctness" can equally well be captured by the following formula:

\[ ∀s' ∈ \text{Dom}(G): (Sns'm(c)) ≤ s'G. \]  (3.2)

The invariant assertion method to prove concurrent programs states (see e.g. [DLJ78]) that an assertion is an invariant over the entire program if it is an invariant over the program's atomic actions. This can be given the following formal justification. Call an assertion P: S+{true,false} an "invariant over c" iff

\[ ∀s', s ∈ S: (P(s') ∧ (s', s) ∈ m(c)) \Rightarrow P(s). \]  (3.3)

Proposition 3.1: If P is an invariant over all a ∈ A then P is an invariant over c.

Proof: Under the assumptions given, the truth of P "propagates" through all execution sequences (2.8).

3.4 Effect-Replaceability

Replacing an atomic action a ∈ A by an effect-equivalent action a' does not alter the semantics of c. More precisely, let c[a→a'] denote a copy of c in which a is replaced by a' (outer brackets remaining). Then the following holds.

Proposition 3.2: m(a) = m(a') \Rightarrow m(c[a→a']) = m(c).

Proof: In the definition of m(c) the atomic actions a ∈ A enter only in the form m(a) in (2.9).

Conversely, let "x" denote any syntactic portion of c which could, syntactically speaking, be enclosed within atomic action brackets, i.e. for which c[x→<x>] is again a concurrent program.

Proposition 3.3: If ∀y: (m(y) = m(x) ⇒ m(c[x→y]) = m(c))

then m(c[x→<x>]) = m(c).

Proof: m(x) = m(<x>).

Hence the property of effect-replaceability expressed in proposition 3.2 can be interpreted as the characteristic property of atomic actions.
3.5 Fairness

Our fairness property (2.16) has been introduced as a generalization of the maximality property (2.15). It forces, via (2.17), programs such as P6 to terminate. P6 thus becomes a program of unbounded non-determinacy [DIJ76] whose weakest precondition function is not continuous. Since our and other work (e.g. [BAC80]) seems to indicate that one can live without the continuity of the wp, we do not insist on the latter. Even without the continuity of the wp we have consistency of semantics as will be shown in propositions 3.4 and A.1-A.5. Besides, in exchange for the continuity of the wp, one gains maximality which also seems to be a "nice" property.

Our fairness postulate is somewhat at variance with the usual [DIJ76] "erratic" interpretation of non-deterministic commands. For example, consider P9†.

```
    do x:=0 + x:=1 OR y:=y+1 od

Program P9
```

The erratic interpretation means that there is no rule whatsoever influencing the decision between the two alternatives in P9. In particular, y:=y+1 can be executed infinitely often, whence P9 may fail to terminate.

We compare P9 with the following two programs.

```
    do <x:=O> + <x:=1 OR y:=y+1> od

Program P10
```

```
    do <x:=0> + <x:=1> OR <y:=y+1> od

Program P11
```

Because in P10 the body of the loop is a single atomic action our fairness property does not apply, still allowing P10 to fail to terminate. By contrast, for P11 the sequence in which y:=y+1 is always executed is valid but not maximal, whence (2.16)-(2.17) require P11 to terminate.

We leave open whether or not this should be considered a desirable state of affairs. On the one hand, a case can be made that P10 and P11 are different, in that the former represents "internal" choice within a single action while the latter represents "external" choice between actions. On the other hand, the fact is somewhat inconvenient that atomic action brackets cannot unthinkingly be omitted in a sequential component even if it contains only local variables. Conditions under which they can be omitted are however given in the next section (axiom (3.4)).

The fairness property has been studied more closely in recent research, e.g. in [PAR80,APT81a,MAN81]. Our definition corresponds to what has been called "strong fairness" in [APT81a]. It is beyond the scope of the present paper either to examine more closely the arguments for or against (2.16)-(2.17), or to analyse the exact relationship between this and other recent definitions.

3.6 Consistency With Weakest Precondition Semantics

In this section we show that, except for situations such as illustrated by P9-P11, our semantics includes the wp semantics of [DIJ76] as a special case. To this end, let \( \tilde{c} \) denote a SEQPROG which does not contain the parallel operator \( || \), and let \( c \) be derived from \( \tilde{c} \) by removing all atomic action brackets.

**Lemma 3.1:** \( c \) is a guarded command program (SPROG) as defined in Appendix A. Conversely, every SPROG \( c \) can be derived from a suitable SEQPROG \( \tilde{c} \) in this way.

**Proof:** The relevant parts of (SYN1)-(SYN8) are just a redundant version of the SPROG syntax defined in Appendix A. □L3.1

Let \( \tilde{c} \) be called "erratic" iff it satisfies the following.

Every infinite execution of \( \tilde{c} \) is maximal. (3.4)

†Henceforth we abbreviate "if true->c_1 [true->c_2 fi]" to "c_1 OR c_2".
(Maximality in the sense of (2.16) is meant.) By (3.4) we exclude programs such as P11. We then have:

**Proposition 3.4:** Let \( \mathcal{C} \) be erratic. Let \( c \) be derived from \( \mathcal{C} \) by omitting atomic action brackets and let \( m_0(c) \) as in Appendix A. Then

\[
\forall s', s \in S: (s', s) \in m_0(c) \Rightarrow (s', s) \in m(\mathcal{C}) \land \text{is}\text{'s} m(\mathcal{C}).
\]  

(3.5)

**Proof:** See Appendix B. \( \square \)

Since (as shown in Appendix A) \( m_0(c) \) is equivalent to \( wp(c) \), proposition 3.4 also states the precise relationship between \( m(\mathcal{C}) \) and \( wp(c) \). In particular, one can retrieve \( wp(c) \) from \( m(\mathcal{C}) \) by using the direction \( \Rightarrow \) of proposition 3.4. Conversely, \( m(\mathcal{C}) \) cannot in general be retrieved from \( wp(c) \), the reason being that, for example, the \( m(\mathcal{C}) \) semantics can distinguish between the two programs "skip OR abort" and "abort" while neither the \( m_0 \) semantics nor the \( wp \) semantics can.

**Remark:** The latter issue can be cleared up in general. Relations over \( S \times S \), such as \( m_0(c) \) (see (A.16) in Appendix A), are too poor to distinguish between the three programs "skip", "skip OR abort" and "abort". The second program has to be equated either to "skip" (as done, for example, in [HAR79], page 8) or to "abort" (as in [DJF76]). The possibility of distinguishing between the three programs "skip", "skip OR abort" and "abort", and hence a more satisfactory treatment of non-determinacy, comes with the addition of \( 1 \) to the state space, such as in (2.2).

(End of Remark.)

### 3.7 Consistency With the Owicki-Gries Calculus

Let us call the construct \( \{P\}_c(Q) \), where \( P, Q : S \rightarrow \{true, false\} \) are unary predicates, a "partial correctness statement" (p.c.s.) over \( c \) iff

\[
\forall s', s \in S: (P s') \land (s', s) \in m(c) \Rightarrow Q(s).
\]  

(3.6)

In [OW176a], Owicki and Gries have defined a number of conditions which allow one to derive from a set of individual p.c.s. \( \{P_i c_i Q_i\} \) of the sequential components the combined p.c.s. \( \{P_1 \land \ldots \land P_n c_1 \mid \ldots \mid c_n (Q_1 \land \ldots \land Q_n)\} \). Their calculus (OGC for short) is slightly adapted in [BES82] to fit our framework, whereafter the following result can be proved:

**Proposition 3.5:** If \( \{P\}_c(Q) \) can be deduced in OGC then it is a p.c.s.

Conversely, if \( \{P\}_c(Q) \) is a p.c.s. then it can be deduced in OGC (with the help of auxiliary variables).

**Proof:** See [BES82]. \( \square \)

**Proposition 3.5** re-states the soundness and completeness of OGC, for which see also [OW176b] and [APT81b]. This guarantees that, as far as statements of partial correctness are concerned, the control sequence approach and OGC are equivalent in principle. Nevertheless, the respective proof methods differ. As a rule, less auxiliary variables are needed in the former method. However, as will be shown in section 4, the two methods can well be used in combination.

### 3.8 Execution Sequences

In this section we explain our perhaps somewhat unusual definition of an execution sequence as an alternation between states and atomic actions. This approach has, in fact, been inspired by net theory [PET80] where alternation between state elements and transition elements is one of the fundamental concepts. Our aim is to explain why we use an approach which is different from other proposed definitions of execution sequences (or histories), such as, for example, strings of actions in [HOA78a], strings of actions and strings of vectors of actions in [LAI79], strings of states (including control states) in [MAN81], or strings of states with a component indicator in [APT81b].

The question may be asked, why would it not be sufficient to consider just sequences of actions? In our context, the answer is that the inclusion of states eases the treatment of internal non-determinacy within atomic actions. For example, consider P12 on the next page.
\( <x:=0 \text{ OR } x:=1> \)

**Program P12**

We can distinguish between the two executions \((x:=0)P12(x:=0)\) and \((x:=0)P12(x:=1)\) whose projections onto P12 are, however, equal. This example can easily be modified to a program with the property that there are two different execution sequences whose atomic actions are the same ones in the same order, one of which is maximal in the sense of (2.16) while the other one is not. These matters would be more difficult to express without intermediate states.

On the other hand, why not use just sequences of states? Again, the answer is that in our model the appearance of actions in (2.8) eases the definition of control by allowing the relevant projections to be formed in the first place. Considering just sequences of variable states suffices for sequential programs (see (A.22)-(A.24)) but not for concurrent programs where control must enter. In [APT81b] this is achieved by component indicators while [MAN81] uses control states. It must be left to the reader to judge upon the relative merits of either approach.

**Remark:** Here we only point out that our semantics is inequivalent to the one given in [APT81b]. This is so, firstly, because in [APT81b] relations over \( S \times S \) are used, for which compare the remark at the end of section 3.6. Secondly, both nesting and non-determinacy within sequential components are disallowed in [APT81b], and it appears that the component indicator method cannot easily be extended to the more general programs considered in the present context.

```latex
\textbf{if} <true \Rightarrow x:=x+1> \textbf{fi} \textbf{if} <true \Rightarrow x:=x+1>; <z:=x> \textbf{fi}
```

**Program P13**

For instance, in P13, how is one to distinguish between control residing after the first alternative and control residing in the middle of the second alternative?

(End of remark.)

Apart from this, alternating sequences offer a practical advantage for proofs, in that it becomes possible to determine "the state immediately preceding" a given action occurrence. For example, if \( s \) precedes \( a \) in \( u \) then the guard of \( a \) must have been true in \( s \), regardless of any worries about whether \( s \) could perhaps have been changed by other components; this becomes important, for example, in section 4.2. In this way, the notion of a "global state" presents no difficulty, nor does the notion of a "shared variable".

Finally, a comment on "interleaving" versus "concurrency". Although we consider only sequential executions, this should not be taken to mean that a concurrent program \( c \) has to be executed sequentially. Rather, any execution of \( c \) (in particular, a concurrent one!) should be regarded as admissible, as long as the formal semantics of \( c \) is adhered to. The set of admissible executions has been characterised in [BES81c].

4. **EXAMPLES**

The examples in this section are intended to show some applications of the control sequence model in formal proofs of concurrent programs. The examples have been chosen relatively non-standard so that, it is hoped, they are also interesting in themselves. Sections 4.1 and 4.2 contain comparisons of alternative assertional and control sequence proofs. Sections 4.3 and 4.4, on the other hand, show how these different styles can be used together in a complementary way in a single proof. In our examples we feel free to extend our notation somewhat; however, no semantic difficulties should arise. The author did not, so far, succeed in finding a non-trivial example featuring the proper nesting of atomic actions in a natural and convincing way.
4.1 Parallel Incrementation

We consider the following program:

\[ \langle x := x+1 \rangle \ | \ | \langle x := x+2 \rangle \]

Program P14:

Starting with \( x = x' \) there are two possible execution sequences leading to the same final state. Hence, in binary relation notation (for which compare [BES81b]),

\[ m(P14) = (x = x' + 3). \quad (4.1) \]

This (formal) proof compares favourably with the proof of (4.1) given in [OWI76a] which uses auxiliary variables and is not repeated here.

4.2 A Fixpoint Program

We reconsider a slight variant of the fixpoint program discussed in [DIJ78], retaining however its essence.

\[
\text{const } N > 0; \\
\text{var } e, h: \text{array } [0..N-1] \text{ of } 0..1; \\
(\text{initially } h_i = 1, e_i \text{ arbitrary for } 0 \leq i \leq N-1) \\
P15 = c_0 \ | \ ... \ | \ c_{N-1}, \text{where for } 0 \leq i \leq N-1, \\
c_i = \text{do} \exists j: h_j = 1 > + \\
a1 \quad \text{if } < e_i = 1 + h_i := 0 > \\
a2 \quad \square < e_i = 0 + \forall j: (e_j = 0 \text{ OR } (e_j = 1)>; \\
a2j \quad \forall j: < h_j = 1 > \\
\text{fi}
\]

Program P15

We wish to prove that the following is an invariant:

\[ (\exists j: h_j = 1) \lor (\forall i: e_i = 1). \quad (4.2) \]

To translate P15 into the program given in [DIJ78], put \( e_i = 1 \Leftrightarrow y_i = f_i(y) \); (4.2) then means that on termination, \( y \) is a fixpoint of \( f \).

Dijkstra's assertional proof [DIJ78].

Introduce auxiliary variables \( s_{ij} \) as follows.

\[
\text{var } s: \text{array } [0..N-1] \times [0..N-1] \text{ of } 0..1; \text{ (initially } s_{ij} = 1) \\
c_i = \text{do} \exists j: h_j = 1 > + \\
a1 \quad \text{if } < e_i = 1 + h_i := 0 > \\
a2 \quad \square < e_i = 0 + \forall j: ((e_j = 0 \text{ OR } (e_j = 1); s_{ij} := 0)>; \\
a2j \quad \forall j: < h_j = 1 >; s_{ij} := 1 > \\
\text{fi}
\]

Program P15'

Define \( \text{var } R: \text{array } [0..N-1] \text{ of } 0..1 \) as the minimal solution of (4.3) and (4.4).

\[ \forall j: (h_j = 0) \lor (R_j = 1) \quad (4.3) \]
\[ \forall i, j: (R_i = 0) \lor (s_{ij} = 1) \lor (R_j = 1). \quad (4.4) \]

Then, by applying proposition 3.1, prove the following invariant.

\[ \forall i: (e_i = 1) \lor (R_i = 1). \quad (4.5) \]

(This is the intricate bit of the proof, omitted here for the sake of brevity.)
This completes the assertional proof, since all $h_i = 0$ implies (by def.) all $R_i = 0$, which by (4.5) implies all $e_i = 1$. Different derivations of this proof are described in [DIJ78] and in [BES79].

A control sequence proof.

We prove (4.2) by contradiction without the use of auxiliary variables. Suppose that there exists a sequence $u = s_0 a_1 \ldots a_s s_r$ of the form (2.8) satisfying both (2.9) and (2.10), and that in $s_r$ (4.2) is violated. In particular, there exists a component $c_{i_0}$ such that $e_{i_0} = 0$ and $h_{i_0} = 0$ in $s_r$. We construct a contradiction by working up backward the sequence $u$.

Because $h_{i_0} = 0$ in $s_r$ and because $h_{i_0} = 1$ to start with, there must be an index $k$, $1 \leq k \leq r$, such that $a_k$ is an execution of $a_1$ (see P15) for $c_{i_0}$. Define $p_0$ as maximal amongst these indices $k$.

Because $e_{i_0} = 0$ in $s_r$ there exists a component $c_{i_1}$ (possibly $i_0 = i_1$) such that some $a_{q_0}$, with $p_0 < q_0$, is an execution of $a_2$ for $c_{i_1}$.

We now consider the state $s_{q_0-1}$, i.e. the state just before $a_{q_0}$. We know that $e_{i_1} = 0$ in this state because of the guard of $s_{q_0}$. Suppose that $h_{i_1} = 1$ in $s_{q_0-1}$. In order to switch $h_{i_1}$ to 0 in $s_r$, all of the $N$ atomic actions $a_{2j}$ of $c_{i_1}$ must occur after $a_{q_0}$ (otherwise (2.10) would not be true for $u$). In particular, $h_{i_0}$ is set to 1 by this, which contradicts the maximality of $p_0$.

Therefore, in the state $s_{q_0-1}$ we have both $e_{i_1} = 0$ and $h_{i_1} = 0$.

We can now repeat this argument to show the existence of indices $p_1$ and $q_1$ ($p_1 < q_1 < q_0$) and a component $c_{i_2}$, such that both $e_{i_2} = 0$ and $h_{i_2} = 0$ in the state $s_{q_1-1}$ (the assumption $h_{i_2} = 1$ in $s_{q_1-1}$ can be shown to contradict either the maximality of $p_0$ or the maximality of $p_1$).

In sum, we construct a never-ending descending sequence of indices $... q_2 < q_1 < q_0$ in $u$, in contradiction to its finite length.

Short discussion

The main difficulty of the proof of P15 is that "between a2 and a2j" things would almost go wrong, were it not for the fact that all $h_i$ are set to 1 in a2j. This is expressed by the $s_{i,j}$ in the assertional proof, and by an application of property (2.10) in the control sequence proof. Thus in this case the two methods lead to quite different proofs. It is not asserted that the latter proof is comparable in elegance to the former. However, it is claimed that the two proofs are comparably complex and efficient. The author feels satisfied with the control sequence proof because it formalises an "operational" argument which he did not succeed in making formal when first studying the program.

4.3 A Parallel Warshall Algorithm

Let a finite directed graph be given whose edges are weighted with positive numbers. The problem to be solved is to compute the minimal distances between vertices. Formally, let $V$ be a finite set of vertices and let $E \subseteq V \times V$ be a set of directed edges. Let $\omega: E \to \mathbb{N}$ be an edge weight function. Let $w = (v_0, v_1), \ldots, (v_{m-1}, v_m)$ ($v_j \in V$ for $0 \leq j \leq m$) be a path from $v_0$ to $v_m$. Then

$$1(w) = \sum_{j=0}^{m-1} \omega(v_j, v_{j+1})$$

(4.6)

is called the "length" of $w$. By $1_{\min}(v', v)$ we denote the minimal length of a path from $v' \in V$ to $v \in V$.

We define two matrices, $W$ and $L$, as follows.

$$W: V \times V \to \mathbb{N}_0(\infty), \quad W_{v', v} = \begin{cases} \omega(v', v) & \text{if } (v', v) \in E \\ \infty & \text{otherwise} \end{cases}$$

(4.7)

$$L: V \times V \to \mathbb{N}_0(\infty), \quad L_{v', v} = \begin{cases} 1_{\min}(v', v) & \text{if } \exists \text{path from } v' \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

(4.8)
Warshall's algorithm [WAR61] iteratively transforms \( W \) into \( L \) by considering, for all \( z \in V \), whether or not, using \( z \), a shorter path can be found from \( x \in V \) to \( y \in V \). Thus the following guarded command lies at the heart of the algorithm:

\[
W_{xz} + W_{zy} < W_{xy} \quad \text{or} \quad W_{xy} := W_{xz} + W_{zy}.
\]  \hfill (4.9)

The ordering of (4.9) is not arbitrary. However we shall prove the following program correct:

\[
\text{do for all } z \in V + \\
\quad a_1 \quad \text{if } W_{xz} + W_{zy} < W_{xy} \quad \text{then } W_{xy} := W_{xz} + W_{zy} \quad \text{else } \text{ skip} \\
\quad a_2 \quad \text{fi} \\
\text{od}
\]

Program P16.

We conduct the proof by first introducing auxiliary variables and then proving an invariant using the control sequence model. For each \((x,y) \in V^2\) we introduce a set \( V_{xy} \subseteq V \) as follows.

\[
\text{do for all } (x,y) \in V^2 + V_{xy} := (x,y) \text{ od}; \\
\text{do for all } z \in V + \\
\quad a_1 \quad \text{if } W_{xz} + W_{zy} < W_{xy} \quad \text{then } W_{xy} := W_{xz} + W_{zy}; \quad V_{xy} := V_{xy} \cup \{z\} \\
\quad a_2 \quad \text{else } V_{xy} := V_{xy} \cup \{z\} \\
\quad \text{fi} \\
\text{od}
\]

Program P16'.

First we note that the program terminates since \( V \) is finite. Then we note that on termination, \( V_{xy} = V \) for all \( x,y \), since either \( a_1 \) or \( a_2 \) is executed for all triples \((x,y,z) \in V^3\). Finally we prove the following invariant:

\[ W_{xy} \text{ indicates the minimal length of a path from } x \text{ to } y \text{ using only vertices from } V_{xy}. \] \hfill (4.10)

This completes the proof since on termination, (4.10) implies \( W = L \).

Proof of (4.10):

A general execution of P16' is \( u = s_0 a_1 s_1 a_2 \ldots a_r \) where \( a_1 \) is an execution of \( a_0 \) and \( a_2, \ldots, a_r \) are executions of either \( a_1 \) or \( a_2 \). By definition of \( W \) (cf. (4.7)), (4.10) holds in \( s_1 \). Suppose (4.10) holds in \( s_{r-1} \); we prove that it holds in \( s_r \). We denote by \( V'_{xy} \) ("dashed"), etc., the values of the variables in \( s_{r-1} \), and by \( V_{xy} \) ("undashed"), etc., the values of the variables in \( s_r \). \( W.L.O.G. \), assume \( x \neq z \).

Case 1: \( a_r = a_1 \)

Then \( V_{xy} = V'_{xy} \cup \{z\} \), \( W_{xy} = W_{xz} + W_{zy} = W'_{xz} + W'_{zy} < W_{xy} \).

Suppose \( w \) is a path from \( x \) to \( y \), using vertices from \( V_{xy} \), whose length \( l(w) \) is smaller than \( W_{xy} \). \( W.L.O.G. \), assume \( w \) does not contain any vertex twice.

If \( w \) does not use vertex \( z \) then the fact that \( l(w) < W_{xy} \) contradicts the hypothesis that (4.10) holds in \( s_{r-1} \).

If \( w \) uses \( z \) then it can be partitioned into a path from \( x \) to \( z \) and another path from \( z \) to \( y \), whose lengths are \( l_{xz} \) and \( l_{zy} \), respectively.

We also have \( V_{xy} \setminus \{y\} \subseteq V_{xz} \).

Proof: let \( z' \in V_{xy} \setminus \{y\} \), \( z' \neq x \), then by (2.10) \( \exists \gamma: 0 < \gamma < r - 1 \) and \( a_r \) is \( a_1 \) or \( a_2 \) for \((x, z, z')\), whence \( z' \in V_{xz} \).

Hence the hypothesis that (4.10) holds in \( s_{r-1} \) allows us to conclude \( l_{xz} \geq W_{xz} \), and analogously \( l_{zy} \geq W_{zy} \).
In all, \( l(w) = l_{xy} + l_{zy} \geq W_{xz} + W_{xy} = W_{xy} > l(w) \), contradiction.

Case 2: \( a_x = a_2 \): Quite analogous; details are left to the reader. \( \square(4.10) \)

The proof clearly shows why the outer loop in P16 cannot be parallelised; otherwise the conclusions \( l(x) \geq W_{xy} \) and \( l(y) \geq W_{xy} \) would no longer be guaranteed. Hence Warshall's algorithm seems to be "sequential of degree \(|V|". Note that P16 contains but the "logical" serialisation; in this case, the parallelisation seems to elucidate rather than to obscure, the essence of the algorithm.

4.4 A Parallel Program Finding Euler Paths

In a connected directed graph which contains for each edge also its opposite edge, Euler cycles always exist. The following is a well-known sequential method for finding an Euler cycle [ORE62]. Starting at an arbitrary vertex one follows edges which have not previously been traversed. If one reaches a vertex for the first time then one marks the entering edge, following its opposite edge only as a last resort. If \( M \) is the number of edges then this sequential algorithm is of time complexity \( O(M) \), as all edges are scanned in sequence. In this section we develop a concurrent algorithm which consists of \( M \) sequential components of constant time.

Suppose that the opposite of an edge \( e \) is given by \( ^{-}e \). Our plan is to deliver an Euler cycle in a function "suc" on edges which gives for each edge its successor edge \( \text{suc}(e) \). We start with

\[
\text{suc}(e)=e
\]

and we choose one of the small suc-cycles which exist by (4.11) as our starting cycle which is to be extended to an Euler cycle. To this end we colour an arbitrary edge \( e_0 \) white and all other edges black \( \text{col}(e_0) = W \) and \( \text{col}(e) = B \) for \( e \neq e_0 \) initially.

The \( e_0 \rightarrow e_0 \) cycle will be extended under the invariance of

There exists a suc-cycle containing exactly all white edges and their opposites. \( \quad (4.12) \)

In our solution (P17 below) a sequential process \( c_e \) is associated with every directed edge \( e \). By \( \text{IN}(\text{head}(e)) \) we denote the set of edges which have the same head vertex as \( e \).

\[
\begin{align*}
\text{c}_e &= \text{if } \exists \text{f} \in \text{IN}(\text{head}(e)) : \text{col}(\text{f}) = W > \text{then} \\
&\text{if } \text{col}(e) = B \rightarrow "\text{choose f}\in\text{IN}(\text{head}(e)):\text{col}(\text{f})=W"; \\
&\quad (\text{suc}(e), \text{suc}(e)) := (-f, -e); \\
&\quad \text{col}(-e) := W > \\
&\text{else} \\
&\quad \text{<col}(e) = W \rightarrow \text{skip}> \\
&\text{fi} \\
&\text{fi} \\
\end{align*}
\]

Program P17

We shall prove the following by means of control sequences.

On termination of P17, \( \text{col}(e) = W \) iff \( \text{col}(-e) = B \). (4.13)

For, suppose that \( \text{col}(e) = \text{col}(-e) = B \) for some edge \( e \). Because the graph is connected, we can assume \( e \) to border on some white edge \( f \); w.l.o.g. we can even assume that \( \text{IN}(\text{head}(e)) \). Then the execution by which this state has been reached is not complete in the sense of (2.11) because the component \( c_e \) could be executed, but has not been executed (otherwise \( \text{col}(-e) = W \)). This finishes the proof, because under the assumption (4.13), the suc-cycle which exists by (4.12) is an Euler cycle.

The proof of P17 follows a pattern which can often be found: there is a partitioning into an invariant (4.12) stating that some property is "always" true, and a termination condition (4.13) stating that another property is "eventually" true. We have shown that, due to the distiction between (2.10) and (2.11), the control sequence formalism is capable of handling both types of property.
5. CONCLUSION

We have developed a relational framework for concurrent programs which can be applied in formal proofs. The basic idea has been that the semantics of a concurrent program should be defined inductively in terms of the relational semantics of its atomic parts. We have established consistency results between this semantics and both sequential non-deterministic semantics and the Owicki-Gries proof method. Furthermore we have conducted several example proofs in order to test the practical applicability of the resulting proof method.

It is perhaps not too daring to assert that the semantics of concurrent programs such as considered in this paper (i.e. shared data programs) is, at the present state of knowledge, almost as well understood as the semantics of sequential non-deterministic programs. The general question of the interplay between relational, operational and assertional semantics for such programs, and the nesting of atomic actions can, in the opinion of the author, on the whole be regarded as settled. The question of fairness, on the other hand, appears to deserve some further attention. However, the author feels sure that the framework given in this paper can prove a sound one in order to formulate and compare various possible definitions of "fairness".

Giving the semantics of a concurrent program by means of an effect relation of the form (2.2) does not lead to all aspects being captured. For example, different kinds of deadlock (partial and total deadlock) cannot be distinguished, nor can different kinds of non-terminating programs. For such distinctions, additional structure would be necessary, for instance, by introducing several different non-termination symbols, or by switching to sets of execution (rather than relations) altogether. While appreciating that such generalisations may be desirable, the author again feels confident that the present framework is flexible enough to be extended in such ways.

An aspect which has been neglected in this paper concerns the actual design of concurrent programs. On the one hand, effect relations such as defined here fit the "goal-oriented" approach to program construction. Moreover it is certainly true that general design heuristics (such as "divide and conquer") apply to concurrent programming just as well as to sequential programming. On the other hand, it would also be desirable to have at one's avail a set of concurrency-specific heuristics; for example: given a problem, how is one to decide how many sequential components the intended solution is to have? Further experience in the design of non-trivial concurrent programs would be helpful towards the goal of obtaining such heuristics.

Connections remain to be explored to other approaches towards concurrent computation which are, at least superficially, substantially different from the shared data model, such as CSP [HOA78b]. In [BES81a] the author has spelt out his belief that, semantically speaking, the shared data model and CSP are not too incompatible. However there seems to be a difference when it comes to practicability and implementation. Solutions which may "look messy" in one model may turn out "nice" in the other one. It is not obvious (and has not been our concern) how the language described in this paper can be implemented. However, this question is, of course, not to be dismissed. The author feels that before implementation it would be wise to subject the notation to a number of embellishments such that useful, but semantically irrelevant, "hints to the compiler" can be given, such as, for example, a shared-local qualification for variables; this would also make a connection to CSP where the shared-local distinction is in-built.
A. APPENDIX: RELATIONAL SEMANTICS AND PREDICATE TRANSFORMERS

This Appendix contains generalisations of a set of results which are by now quite well known [WAN77, PLO80].

A.1 Bijection Between Relations and Predicate Transformers

Let \( S \) be a countable set. Let \( \wp: 2^S \rightarrow 2^S \) be called a "predicate transformer" iff it is strict (A.1) and infinitely multiplicative (A.2).

\[
\wp(\emptyset) = \emptyset \\
\wp(\bigcap_i X_i) = \wp(\bigcap_i X_i) \text{ for every } (X_i | i \in I) \text{ where } I \neq \emptyset \text{ is an index set and } X_i \subseteq S \text{ for all } i \in I.
\]

(A.1)

(A.2)

Let an arbitrary relation \( m_0 \subseteq S \times S \) be given and define \( \wp = \wp(m_0) \) by setting, for all \( X \subseteq S \),

\[
\wp(X) = \{ s' \in S | \emptyset \subseteq s' \subseteq m_0 \subseteq X \}.
\]

(A.3)

Proposition A.1: \( \wp \) as defined by (A.3) is a predicate transformer.

Proof: First we check (A.1): \( \wp(\emptyset) = \{ s' \in S | \emptyset \subseteq s' \subseteq m_0 \subseteq \emptyset \} = \emptyset \).

Then we check (A.2):

\[
s' \in \wp(\bigcap_i X_i) \iff \emptyset \subseteq s' \subseteq m_0 \subseteq \bigcap_i X_i \iff \exists s \in S: (s', s) \in m_0 \cap \bigcap_i X_i \iff \exists s \in S: (s', s) \in m_0 \cap s \in \bigcap_i X_i \iff \forall s \in S: (s', s) \in m_0 \iff s \in \wp(\bigcap_i X_i).
\]

(by A.3)

Conversely, let \( \wp: 2^S \rightarrow 2^S \) be an arbitrary predicate transformer and define \( m_0 = m_0(\wp) \) by setting, for all \( s' \in S \):

\[
s' \in m_0 \iff \{ \emptyset \text{ if } s' \notin \wp(S) \} \subseteq \{ Y \subseteq S | s' \in \wp(Y) \} \text{ if } s' \in \wp(S).
\]

(A.4)

Proposition A.2: (A.3) and (A.4) are two-sided inverses of each other.

Proof:

(i) We prove that applying first (A.3) and then (A.4) gives identity.

Since \( s' \in \wp(S) \iff s' \equiv \emptyset \), we have to prove that for all \( s' \in S \) the following equation holds:

\[
s' \in m_0 = \{ \emptyset \text{ if } s' \equiv \emptyset \} \subseteq \{ Y \subseteq S | s' \in \wp(Y) \} \text{ if } s' \in \wp(S).
\]

(A.4')

The first equality of (A.4') is trivially true. In case \( s' \equiv \emptyset \) it remains to be proved that

\[
s' \in m_0 \iff Y \subseteq S \text{ if } s' \equiv \wp(S).
\]

(A.4'')

(ii) We prove that applying first (A.4) and then (A.3) gives identity.

For an arbitrary set of states \( X \subseteq S \), we have to prove the following equivalence:

\[
\forall s' \in S: s' \in \wp(X) \iff \emptyset \subseteq \{ Y \subseteq S | s' \in \wp(Y) \} \subseteq X \iff \{ Y \subseteq S | s' \in \wp(Y) \} \subseteq X.
\]

(A.3')

Assume first that \( s' \equiv \wp(S) \).

Then \( \wp(S) = \{ Y \subseteq S | s' \equiv Y \} = \emptyset \).

On the other hand, since \( \wp(X) = \wp(S \cap X) = \wp(S) \cap \wp(X) \) by (A.2), it follows that \( \wp(X) = \wp(S) \) as well.

Thus (A.3') has been proved for \( s' \equiv \wp(S) \).
It remains to be proved that
\[ \forall s' \epsilon \text{wp}(S): \text{wp}(X) \Rightarrow s' \epsilon \text{wp}(Y) \subseteq X. \]  
(A.3"")

First we show that \( s' \epsilon \text{wp}(Y) \) for any \( s' \epsilon \text{wp}(S) \).
Assume \( s' \epsilon \text{wp}(Y) \).
Then \( \emptyset = \text{wp}(\emptyset) \)  
\( \text{(by (A.1))} \)
\[ = \text{wp}(\emptyset) \emptyset \text{wp}(Y) \) \( \text{(by assumption)} \)
Since, due to \( s' \epsilon \text{wp}(S) \), the set \( \{Y \mid s' \epsilon \text{wp}(Y)\} \) is not empty, \( (A.2) \) is now applicable, yielding
\( \emptyset = \bigcap \{ \text{wp}(Y) \mid s' \epsilon \text{wp}(Y) \} \).
This is a contradiction because, \( s' \epsilon \bigcap \{ \text{wp}(Y) \mid s' \epsilon \text{wp}(Y) \} \).
Now it remains to be proved that
\[ \forall s' \epsilon \text{wp}(S): \text{wp}(X) \Rightarrow \bigcap \{Y \mid s' \epsilon \text{wp}(Y)\} \subseteq X. \]  
(A.3""")

(\( \Rightarrow \)): Assume \( s' \epsilon \text{wp}(X) \).
Let \( s' \epsilon \bigcap \{Y \mid s' \epsilon \text{wp}(Y)\} \); setting \( Y = X \) it follows that \( s' \epsilon X \).
Hence the r.h.s. of (A.3""") holds true.
(\( \Leftarrow \)): Assume \( \bigcap \{Y \mid s' \epsilon \text{wp}(Y)\} \subseteq X \).
By (A.2) it follows that \( \text{wp}(\bigcap \{Y \mid s' \epsilon \text{wp}(Y)\}) \subseteq \text{wp}(X) \).
As before, \( \{Y \mid s' \epsilon \text{wp}(Y)\} \) is non-empty, and (A.2) gives \( \bigcap \{\text{wp}(Y) \mid s' \epsilon \text{wp}(Y)\} \subseteq \text{wp}(X) \).
Since \( s' \epsilon \bigcap \{\text{wp}(Y) \mid s' \epsilon \text{wp}(Y)\} \), we also have \( s' \epsilon \text{wp}(X) \), i.e. the l.h.s. of (A.3""). \square

Proposition A.2 generalises the result published in [WAN77] in the sense that Wand takes into account only predicate transformers which are finitely multiplicative (A.5) and continuous (A.6).
\[ \text{wp}(X) \cap \text{wp}(Y) = \text{wp}(X \cap Y) \quad \text{for all } X, Y \subseteq S. \]  
(A.5)
\[ \text{wp}(U_{X_1}) = W_{X_1} \quad \text{for every infinite ascending chain } \quad X_0 \subseteq X_1 \subseteq \ldots \text{ of subsets } X_1 \subseteq S. \]  
(A.6)

Our next proposition shows that predicate transformers in the usual sense (i.e. satisfying (A.5) and (A.6)) are also predicate transformers as defined above.

Proposition A.3: Let \( \text{wp}: 2^S \rightarrow 2^S \). If \( \text{wp} \) is finitely multiplicative and continuous then it is also infinitely multiplicative.

Proof: Because \( S \) is countable, it can be written in the form
\[ S = \bigcup_{j=0}^{\infty} S_j, \quad \text{where the } S_j \text{ are such that } S_0 \subseteq S_1 \subseteq S_2 \subseteq \ldots \text{ and } |S_j| < \infty \text{ for all } j. \]
By (A.6), \( \text{wp}(S) = \text{wp}(U_{S_j}) = W_{S_j} \).
Now let \( \{X_i \mid i \in I\} \) be a set of subsets of \( S \)
where \( I \neq \emptyset \) (but possibly, \( I \) is uncountable).
It suffices to prove \( \bigcap \{X_i \mid s' \epsilon \text{wp}(X)\} \subseteq \bigcap \{X_i \mid s' \epsilon \text{wp}(X)\} \), since the other direction of (A.2) is trivial.
\[ s' \epsilon \bigcap \{X_i \mid s' \epsilon \text{wp}(X)\} \Rightarrow \forall i: s' \epsilon \text{wp}(X_i) \]  
\( \wedge s' \epsilon \text{wp}(S) \)  
\( \Rightarrow \forall i: s' \epsilon \text{wp}(X_i) \wedge s' \epsilon \text{wp}(S_j) \) for some \( j \)  
\( \Rightarrow \forall i: s' \epsilon \text{wp}(X_i \cap S_j) \)  
\( \Rightarrow s' \epsilon \bigcap_{i \in I} \text{wp}(X_i \cap S_j) \).
Now since \( S \) is finite, the set \( \{X_i \cap S_j \mid i \in I\} \) is a countable set of finite subsets of \( S \), say \( \{X_{i_k} \mid k \geq 0\} \), where \( |X_{i_k}| < \infty \) for \( k \geq 0 \).
We define \( Y_k = X_{i_k} \cap \ldots \cap X_1 \cap X_0 \cap \ldots \), then \( Y_{i_k} \subseteq Y_{i_k+1} \subseteq \ldots \) and \( |Y_k| < \infty \) for \( k \geq 0 \)
and \( \cap_{k=0}^{\infty} Y_k = \cap_{i \in I} \cap_{j \in J} X_{i_j} \cap \cap_{j \in J} S_j \).
\[ s' \epsilon \bigcap_{i \in I} \text{wp}(X_i \cap S_j) \Rightarrow s' \epsilon \bigcap_{k=0}^{\infty} \text{wp}(Y_k) \]  
\( \text{(by repeated application of (A.5))} \).
It is easy to prove (A.2) for descending chains of finite subsets of $S$, such as $Y_0 \supseteq Y_1 \supseteq \ldots$. Therefore,

\[
\begin{align*}
s' &\in \wp(Y_k) \Rightarrow s' \in \wp(\bigcap_{i \leq 1} (X_i \cap S_j)) \\
&\Rightarrow s' \in \wp(\bigcap_{i \leq 1} X_i) \land s' \in \wp(S_j). \tag{by def. of $Y_k$} \\
&\Rightarrow (A.5) \tag{by (A.5)} \Box{\text{PA.3}}
\end{align*}
\]

Our next result states how special properties of $m_0$ and $wp$ are related to each other. First we define $m_0$ to be finitary iff (A.7) holds and deterministic iff (A.8) holds.

\[
\begin{align*}
\forall s' &\in S: |s'm_0| < \infty \tag{A.7} \\
\forall s' &\in S: |s'm_0| \leq 1. \tag{A.8}
\end{align*}
\]

Define $wp$ to be additive iff the following holds:

\[
wp(UX_i) = Uwp(X_i) \quad \text{for every set of subsets $X_i \subseteq S$.} \tag{A.9}
\]

**Proposition A.4**: Let $m_0$ and $wp$ correspond to each other via (A.3) and (A.4), resp.

(i) $m_0$ is finitary iff $wp$ is continuous.

(ii) $m_0$ is deterministic iff $wp$ is additive.

**Proof**: Left to the interested reader; or consult [BES82]. \Box{\text{PA.4}}

**A.2 Forward and Backward Semantics of Guarded Commands**

In this section we define the backward semantics $wp$ of guarded command programs as in [DIJ76]. Then we define their forward semantics $m_0$ such that it corresponds to the $wp$ semantics via the bijection (A.3) and (A.4), this correspondence being the content of proposition A.5. We use the following syntax for guarded commands, the start symbol being $SPROG$ (and $V,E,B$ as in section 2.1).

\[
\begin{align*}
SPROG &::= EPROG \mid EPROG;SPROG \\
EPROG &::= \text{abort} \mid \text{skip} \mid V:=E \mid \text{IF} \mid \text{DO} \\
\text{IF} &::= \text{if } B \Rightarrow SPROG \ldots \text{fi} \\
\text{DO} &::= \text{do } B \Rightarrow SPROG \text{ od.}
\end{align*}
\]

Let $c$ be a $SPROG$. In preparation of the definition of $wp(c)$, let $X \subseteq S$ and $Q: S \Rightarrow \{\text{true}, \text{false}\}$. Thanks to the equivalence $scX \Rightarrow Q(s)=\text{true}$ for $scS$, subsets $X$ and unary predicates $Q$ can be used interchangeably. Also, for convenience, we write $wp(c,X)$ and $wp(c,Q)$ instead of, respectively, $wp(c)(X)$ and $wp(c)(Q)$.

\[
\begin{align*}
wp(\text{abort},X) &= \emptyset \tag{A.10} \\
wp(\text{skip},X) &= X \tag{A.11} \\
wp(V:=E,Q) &= Q[V:=E], \text{where } Q[V:=E] \text{ is a copy of } Q \text{ in which all free occurrences of } V \text{ are replaced by } E. \tag{A.12}
\end{align*}
\]

\[
\begin{align*}
wp(c_1; c_2, X) &= wp(c_1, wp(c_2, X)) \tag{A.13} \\
\text{Let } IF &= \text{if } B_1 \Rightarrow c_1 \ldots B_m \Rightarrow c_m \text{ fi} \\
wp(\text{IF},X) &= (B_j; B_j) \land (Vj: B_j \Rightarrow wp(c_j, X)) \tag{A.14} \\
\text{Let } DO &= \text{do } B \Rightarrow c \text{ od} \\
wp(\text{DO},X) &= (\bigcup_{i \leq 0} X_i), \text{ where } X_0 = X \cap \text{not } B \land X_{i+1} = (B \Rightarrow wp(c, X_i)) u X_0. \tag{A.15}
\end{align*}
\]

The forward semantics will be defined as a relation

\[
m_0(c) \subseteq S \times S. \tag{A.16}
\]

Let $s' \in S$ denote an initial state.

\[
\begin{align*}
s' &m_0(\text{abort}) = \emptyset \tag{A.17} \\
s' &m_0(\text{skip}) = \{s'\} \tag{A.18} \\
s' &m_0(V:=E) = \{s\}, \text{ where } s \text{ is as in (2.5)} \tag{A.19} \\
(s', s) &\in m_0(c_1;c_2) \Rightarrow s' \in m_0(c_1) \subseteq \text{Dom}(m_0(c_2)) \land (s', s) \in m_0(c_1) \Rightarrow m_0(c_2) \tag{A.20}
\end{align*}
\]

\[\text{This terminology is however somewhat unfortunate because under the } m_0 \text{ semantics (section (A.2)) the program "skip OR abort" would be called "deterministic"; see also the remark at the end of section 3.6.}\]
Let IF and DO as in (A.14) and in (A.15), respectively.
\[(s',s) \in m'_0(IF) \iff Vj: (B(s') \Rightarrow s'_m(c_j) \neq \emptyset)\]  
\[\land \exists j: (B(s') \land (s',s) \in m'_0(c_j))\]  
For the forward definition of DO we call a sequence of states \(s_0, \ldots, s_r\) "valid" iff
\[Vj, 0 \leq j \leq r-1: B(s_j) \land (s_j, s_{j+1}) \in m_0(c).\]  
(A.22)

Analogously, call an infinite sequence \(s_0, s_1, \ldots\) "valid" iff
\[Vj, 0 \leq j: B(s_j) \land (s_j, s_{j+1}) \in m_0(c).\]  
(A.23)

(A.22) and (A.23) are the analogs of properties (2.9), (2.10) and (2.13).

\[(s',s) \in m_0(DO) \iff \exists \text{ valid sequence } s_0, \ldots, s_r:\]
\[s' = s_0, s_r = s, \text{ and not } B(s_r);\]
\[\land (ii) \text{ not valid sequence } s_0, \ldots, s_r: \]
\[s' = s_0, B(s_r), \text{ and } s_r m_0(c) = \emptyset;\]
\[\land (iii) \text{ not valid infinite sequence } s_0, s_1, \ldots: s' = s_0.\]  
(A.24)

Proposition A.5: Let \(c\) be a SPROG and let wp(c) and \(m_0(c)\) as just defined. Then for all \(X \subseteq S:\)
\[wp(c, X) = (s' \in S | c = c m_0(c) \subseteq X).\]  
Proof: Relatively straightforward but rather lengthy; see [BES82]. □PA.5

Proposition A.5 states that wp(c) and \(m_0(c)\) are related to each other via (A.3). Thanks to proposition A.2 it follows that, conversely, \(m_0(c)\) can be expressed in terms of wp(c) via (A.4).

B. APPENDIX: ON THE PROOF OF PROPOSITION 3.4

As in section 3.6, let \(\tilde{c}\) be a SEQPROG satisfying (3.4) and let \(c\) be derived from \(\tilde{c}\) by removing all atomic action brackets. We wish to prove the following:
\[\forall s', s \in S: (s', s) \in m_0(DO) \Rightarrow (s', s) \in m(DO) \land \forall s' \in m(DO).\]  
(3.5)

Since \(c\) is uniquely determinable from \(\tilde{c}\), but not necessarily vice versa, induction should be conducted over the structure of \(\tilde{c}\). Due to the syntax (SYN)-(SYN) a large number of cases have to be distinguished. We choose to prove (3.5) only for the case of a loop (because this is where property (3.4) comes in), claiming that all other cases can be treated quite similarly; for a full proof see [BES82].

Let \(\tilde{D}O = \text{do } \langle B> \Rightarrow \tilde{c} \mid \text{od}, DO = \text{do } B+c \mid \text{od}.

We prove the following separately:
\[\forall s', s \in S: (s', s) \in m_0(DO) \Rightarrow (s', s) \in m(DO) \land \forall s' \in m(DO).\]  
(B.1)

\[\forall s', s \in S: (s', s) \in m(DO) \land \forall s' \in m(DO) \Rightarrow (s', s) \in m_0(DO).\]  
(B.2)

Our induction hypothesis is that (3.5) already holds for the bodies \(\tilde{c}, c\) of the loop. The proof uses some (hopefully) understandable abbreviations.

Proof of (B.1)
\[\text{if } (s', s) \in m_0(DO)\]
\[\Rightarrow \exists \text{ valid sequence } s_0, \ldots, s_r \text{ with } s' = s_0, s_r = s, \text{ not } B(s_r),\]  
(i)
\[\land \text{ not valid sequence } s_0, \ldots, s_r \text{ with } s' = s_0, B(s_r), s_r m_0(c) = \emptyset.\]  
(ii)
\[\land \text{ not valid sequence } s_0, s_1, \ldots \text{ with } s' = s_0.\]  
(iii)
\[\Rightarrow \text{(by the definition (A.22)-(A.23) of "valid" and by induction hypothesis)}\]
\[\Rightarrow \exists s_0, \ldots, s_r:\]
\[B(s_0), \ldots, B(s_r-1), \text{ not } B(s_r),\]  
(i)
\[(s_j, s_{j+1}) \in m(c) \land \forall s' \in m(c) \text{ for } 0 \leq j \leq r-1;\]  
(ii)
\[s' = s_0 \text{ and } s_r = s;\]  
(iii)
\[\text{not } \exists s_0, \ldots, s_r:\]
\[B(s_0), \ldots, B(s_r)\]  
(i)
\[\text{as (ii)};\]  
(ii)
\[s' = s_0 \text{ and } s_r m_0(c) = \emptyset.\]  
(iii)
First we show that (ii)-(i3) imply \((s',s)\in m(\overline{D})\).

Let \(j\) be such that \(0\leq j \leq r-1\).

By (i2), \((s_j,s_{j+1})\in m(\overline{c})\),
whence \(\exists u_j: s_ju_js_{j+1}\) a complete execution of \(\overline{c}\).

In particular, \(\text{proj}(\overline{c},u_j)\) is a c.c.s. of \(\overline{c}\).

The \(u_j\) can be concatenated as follows to give a larger execution:

\[
u = s_0s_0u_0s_1s_1s_2s_3 \ldots s_{r-1}s_{r-1}u_{r-1}s_r
\]

\(u\) satisfies (2.9) because of (ii) and because the \(u_j\) do so individually.

Further, \(\text{proj}(\overline{D},u)\) is a c.c.s. of \(\overline{D}\) by definition.

By (i3), \(s_0=s'\) and \(s_r=s\).

Hence \((s',s)\in m(\overline{D})\) by (2.12).

It remains to be proved that \(\not\in s'm(\overline{D})\).

Assume the contrary, i.e. \(\not\in s'm(\overline{D})\).

Then (2.17) applies, giving either (a) \(\exists u = s_0a_1 \ldots a_r s_r\) with \(s'=s_0\) and \(s_r=s\),
or (b) \(\exists u = s_0a_1 \ldots a_r s_r\) maximal but not complete
or (c) \(\exists u = s_0a_1 \ldots \) maximal infinite execution.

Suppose (a) holds.

Since \(a_1 \ldots s_r\) is a c.c.s. of \(\overline{D}\) it must be of the form

\[
<B \leq w_1 < B \ldots < B \leq w_q
\]

where \(w_1, \ldots, w_{q-1}\) are c.c.s. of \(\overline{c}\) and \(w_q\) is a c.c.s. of \(\overline{c}\).

Let \(s\) be the state in \(s_0a_1 \ldots a_r s_r\) between \(B < B\) and \(w_q\).

The existence of \(w_q\) shows that \((s_0, s)\in m(\overline{c})\).

By induction hypothesis, \(s_0m(\overline{c}) = \emptyset\), contradicting (ii).

By an entirely similar argument,
(b) and (c) can be brought to a contradiction to (ii) and (iii). \(\Box(\overline{B},1)

**Proof of (B.2)**

\((s',s)\in m(\overline{D})\)

\[\Rightarrow \exists u: s'u' \land \not\in B(s) \land \text{proj}(\overline{D},u)=<B \leq w_1 < B \ldots < B \leq w_r < \not\in B >\text{ for some } r \geq 0,
\]

such that every \(w_j\) \(1 \leq j \leq r\) is a c.c.s. of \(\overline{c}\).

Thus, \(u\) has the following form:

\[
u = s_0s_0w_1s_1s_2s_3 \ldots s_{r-1}s_{r-1}w_rs_r < B > s_r
\]

where \(w_j\) as above and \(s' = s_0, s_r = s\).

If for some \(j\) \((0 \leq j < r)\), \(\not\in s_j m(\overline{c})\) then this contradicts \(\not\in s'm(\overline{D})\).

Hence the induction hypothesis can be applied \(r\) times,
giving \((s_j, s_{j+1})\in m(\overline{c})\) for \(0 \leq j < r-1\).

In other words, the \(s_0, \ldots, s_r\) form a valid sequence of the form (A.24)(i),

It remains to prove (A.24)(ii) and (A.24)(iii).

Assume there exists a valid sequence \(s_0, \ldots, s_r\)
with \(B(s_r)\), \(s'_0=s_0\) and \(s_0m(\overline{c})=\emptyset\).

By induction hypothesis, \((s_j, s_{j+1})\in m(\overline{c})\) for \(0 \leq j \leq r-1\).

By induction hypothesis, \(s_r m(\overline{c})=\emptyset\) implies further that \(\not\in s_r m(\overline{c})\).

Because \(B(s_r)\), we can therefore construct an execution
of \(\overline{D}\) leading from \(s'\) to \(s\).

This contradicts \(\not\in s'm(\overline{D})\), establishing (A.24)(ii).

Finally, assume there exists a valid sequence \(s_0, s_1, \ldots\) of the form
(A.24)(iii), i.e. satisfying \(s'_0=s_0\).

By induction hypothesis, \((s_j, s_{j+1})\in m(\overline{c})\) \(\not\in s_j m(\overline{c})\) for \(0 \leq j\).
Again, this means that
\[ \exists u_j : s_j u_j s_{j+1} \text{ for } 0 \leq j \text{ and } u_j \text{ is a complete execution of } \overline{c}. \]
We define the sequence
\[ u = s_0 < B > s_0 u_0 s_1 < B > s_1 \ldots \]
and claim that \( u \) is an execution of \( D_0 \); this is so, indeed, because \( \text{proj}(D_0, u) \) is a c.s. of \( D_0 \) and because \( u \) satisfies (2.9) by assumption.

Being an infinite execution, \( u \) is maximal by axiom (3.4).

Hence (2.17) applies, giving \( \exists s \in s'm(D_0) \), contradicting the assumption. Thus no valid sequence \( s_0, s_1, \ldots \) with the above properties exists, which establishes \((A.24)(iii)\). □(B.2)□P3.4

C. APPENDIX: SOME NOTATION

\( \in \quad \text{element of} \)
\( \notin \quad \text{not element of} \)
\( \subseteq \quad \text{subset of} \)
\( \subset \quad \text{proper subset of} \)
\( \times \quad \text{Cartesian product} \)
\( \emptyset \quad \text{empty set} \)
\( \cup \quad \text{union} \)
\( \cap \quad \text{intersection} \)
\( 2^S \quad \text{powerset of } S \)
\( |X| \quad \text{cardinality of } X \)

\[ \forall \quad \text{logical or} \]
\[ \exists \quad \text{existential quantifier} \]
\[ \land \quad \text{logical and} \]
\[ \lor \quad \text{universal quantifier} \]
\[ \lnot \quad \text{negation} \]
\[ \Rightarrow \quad \text{implication} \]
\[ \Leftrightarrow \quad \text{equivalence} \]

\[ \square \quad \text{end of proof/proposition/lemma} \]

Operator precedence is as usual, e.g., \( \in \) binds stronger than \( \land \).

For \( m, m_1, m_2 \in S \times S \), \( s, s' \in S \):

\[ s \cdot m = \{ s \in S | (s', s) \in m \} \]
\[ \text{Dom}(m) = \{ s' \in S | s \cdot m = \emptyset \} \]
\[ m_1 \circ m_2 \quad \text{relational composition} \]

\( f : X \rightarrow Y \quad f \text{ is a function from } X \text{ to } Y \)
\( < > \quad \text{atomic action brackets} \)
\( < = \leq \quad \text{less, greater, less or equal, greater or equal, respectively} \)
\( N \quad \text{natural numbers} \)

ACKNOWLEDGEMENTS

The author wishes to express his sincere gratitude to his former colleagues at the University of Newcastle upon Tyne where the major part of this research was being carried out. Special thanks are due to Peter Lauer and Brian Randell for their help and inspiration. This work also owes much to discussions with Mike Shields and Flaviu Cristian. The financial support of the Science Research Council of Great Britain is gratefully acknowledged.

REFERENCES


