PROVING ASSERTIONS ABOUT PARALLEL PROGRAMS

by

E.A. Ashcroft

Research Report CS-73-01

Department of Applied Analysis and= Computer Science

University of Waterloo
Waterloo, Ontario, Canada
January 1973
(revised April 1974)

Faculty of Mathematics

University of Waterloo

Waterloo, Ontario

Canada

Department of Applied Analysis

&

Computer Science

PROVING ASSERTIONS ABOUT PARALLEL PROGRAMS

by

E.A. Ashcroft

Research Report CS-73-01

Department of Applied Analysis and=
Computer Science

University of Waterloo
Waterloo, Ontario, Canada

January 1973
(revised April 1974)

PROVING ASSERTIONS ABOUT PARALLEL PROGRAMS

by

E. A. Ashcroft
Computer Science
University of Waterloo
Waterloo, Ontario
Canada

Abstract

A simple but general parallel programming language is considered. The semantics of programs are defined in a concise and natural way using relations. 'Verification conditions' derived from the semantic definitions enable Floyd's method of proving correctness to be applied the parallel programs. Proofs of properties of programs using the verification conditions are claimed to be more systematic versions of the informal arguments normally used to check parallel programs. A program simulating an elementary airline reservation system is given, and several properties of the program are demonstrated using the technique.

This research was supported by the National Research Council of Canada.

INTRODUCTION

The method of Floyd [2] (see also [6]) for proving assertions about programs has not yet become a widely used technique. Part of the reason for this may be that the method requires a discipline of thought that programmers may find both unfamiliar and unnecessary. For the simple programs on which the method is usually demonstrated, a sceptical programmer could maintain that he finds the proofs harder to understand than the original programs.

The situation changes however when we consider programs in languages of greater semantic complexity. (Complex programs in simple languages probably just result in more complex proofs, and the situation remains essentially the same.) For example recursive programming requires a more 'inductive' type of reasoning than does iterative programming, and producing correctness proofs requires a little clerical effort rather than a mental leap (see [7]).

Parallel programming also requires a different type of reasoning; one can rely much less on the 'obvious'. Since such programs include operating systems, airline reservation systems and the like, it is crucial that this reasoning be correct. Parallel programs are difficult to debug, and the 'proof by test cases' approach is even more unreliable than for sequential programs. The programmer's confidence in his program has to come from very careful analysis of possible situations. Any proof method that ensures that he has considered all eventualities can only be helpful (provided it is not impossibly tedious).

The adaptation of the assertion method that we present here results in proofs that are not unnecessarily tedious, and that follow the sort of reasoning that the programmer would ordinarily have to make. However, the method is not based on such informal reasoning but on a rigourous definition of the semantics of parallel programs. In this respect it differs from other applications of the method of Floyd to parallel programs (see, for example, [5]).

It must be emphasized that the method presented here simply allows one to prove that a program (system) has particular properties. Whether these properties are sufficient to pronounce the system correct or not is outside the scope of this paper. The whole question of what it means for a system to be correct will be considered in a subsequent paper.

PARALLEL PROGRAMS

We wish to keep our parallel programming model as general as possible. All we require is that the model allow a number of computations to be taking place quasi-simultaneously, i.e. the computations can proceed more or less independently but there is some basic level at which no two operations can occur actually simultaneously (or if they do then the effect must be the same as if one preceded the other). Any sort of synchronisation or roadblocking is allowed, and new parallel computations can be initiated and old ones terminated.

No matter how we specify the syntax of such programs, by means of constructs like 'fork' and 'join' or by coroutines or tasking, the specification of the semantics must allow non-determinism. An easy way to do this is in terms of non-deterministic programs, and one is

tempted to take non-deterministic programs as the model, and relegate the usual sorts of parallel programs to the status of convenient syntactic descriptions that are possible in special cases. This was the approach taken in Karp and Miller [4] and to some in extent in Ashcroft and Manna [1]. (In the latter a conventional parallel programming language was considered that used forks and joins, but properties of such programs had to be proved indirectly by way of the corresponding non-deterministic programs.)

Although quite general, the representation of a parallel program by a non-deterministic program has several drawbacks. In particular, the size of the non-deterministic program is some exponential function of the size of the original parallel program. Besides being cumbersome, this means that proofs of correctness tend to be very long. In fact, they are often longer than they need be by an exponential factor. This is because the various quasi-simultaneous computations are usually designed by the programmer to work despite their interactions, and rarely because of the interactions. In other words, enough limitations are imposed, in the form of critical sections and synchronisations, to ensure that each computation can be considered largely independently of the others. The effort of proving n parallel computations correct should then be roughly linear in n rather than exponential, but the latter is the case if we consider the corresponding non-deterministic program.

In this paper we shall specify the syntax of parallel programs using simple fork and join constructions. Synchronisation and

critical section features can be introduced by means of constraints on the execution of statements in particular situations. We shall define the semantics of each parallel program in a natural way in terms of relations. The specification of these relations is directly linked to the program itself and is just as concise. The verification condition used for proving properties of the program is directly obtained from these relations.

Syntax of Parallel Programs

A parallel program P consists of

i) a domain M_P of memory-states. Intuitively, each $m \in M_P$ contains values for all the variables used by the program.

ii) a set of basic operations $G_P = \{g_1, g_2, g_3, \cdots\}$ where $g_i : M_P \to M_P$.

iii) a set of basic tests $Q_P = \{q_1, q_2, q_3, \cdots\}$ where each $q_i \subseteq M_P$ (a relation on M_P) .

iv) a set $L_P = \{L_1, L_2, \cdots\}$ of symbols called labels. The set $C_P = 2^{L_P}$ (the set of all subsets of L_P) will be called the set of control states.

v) a set $S_P = \{s_1, s_2, \cdots\}$ of labelled statements, using G_P and Q_P , with s_1 being the initial statement.

vi) a set of constraints $stops_P$, where a constraint is an element (c, ℓ) of $C_P \times L_P$ such that $\ell \notin c$.

A labelled statement can be of one of the following six forms:

i) L_i : do g_j then go to L_k (operation)

ii) L_i : if q_j then go to L_k else go to L_h (test)

iii) L_i : go to $L_{\alpha_1}, L_{\alpha_2}, \cdots, L_{\alpha_j}$ (fork)

iv) $L_{\beta_1}, L_{\beta_2}, \cdots, L_{\beta_i}$: go to L_j (join)

v) L_i : go to one of $L_{\alpha_1}, L_{\alpha_2}, \cdots, L_{\alpha_j}$ (branch)

vi) L_i : HALT (halt)

where $L_i, L_j, L_k, L_h, L_{\alpha_1}, \cdots, L_{\alpha_j}, L_{\beta_1}, \cdots, L_{\beta_i}$ are labels.

Each label in L_P must occur exactly once labelling some statement in S_P (i.e. before the colon). The initial statement s_1 is labelled with L_1 .

Example: An example program is represented diagrammatically, in Figure 1, with the labels of statements on the edges leading to those statements.

Notation for constraints

It is convenient to regard $\underset{\sim}{\text{stops}}_P$ as a relation on $C_P \times L_P$, and say, for example, $c \underset{\sim}{\text{stops}}_P \ell$. We extend $\underset{\sim}{\text{stops}}_P$ to $C_P \times C_P$: for $c_1, c_2 \in C_P$, $c_1 \underset{\sim}{\text{stops}}_P c_2$ if and only if $c_1 \underset{\sim}{\text{stops}}_P \ell$ for all $\ell \in c_2$.

We also define a relation $\underset{\sim}{\text{stop}}_P$ which extends the first argument of $\underset{\sim}{\text{stops}}_P$ to sets of control states: for $C \subseteq C_P$, $c \in C_P$, $C \underset{\sim}{\text{stop}}_P c$ if and only if $c' \underset{\sim}{\text{stops}}_P c$ for all $c' \in C$.

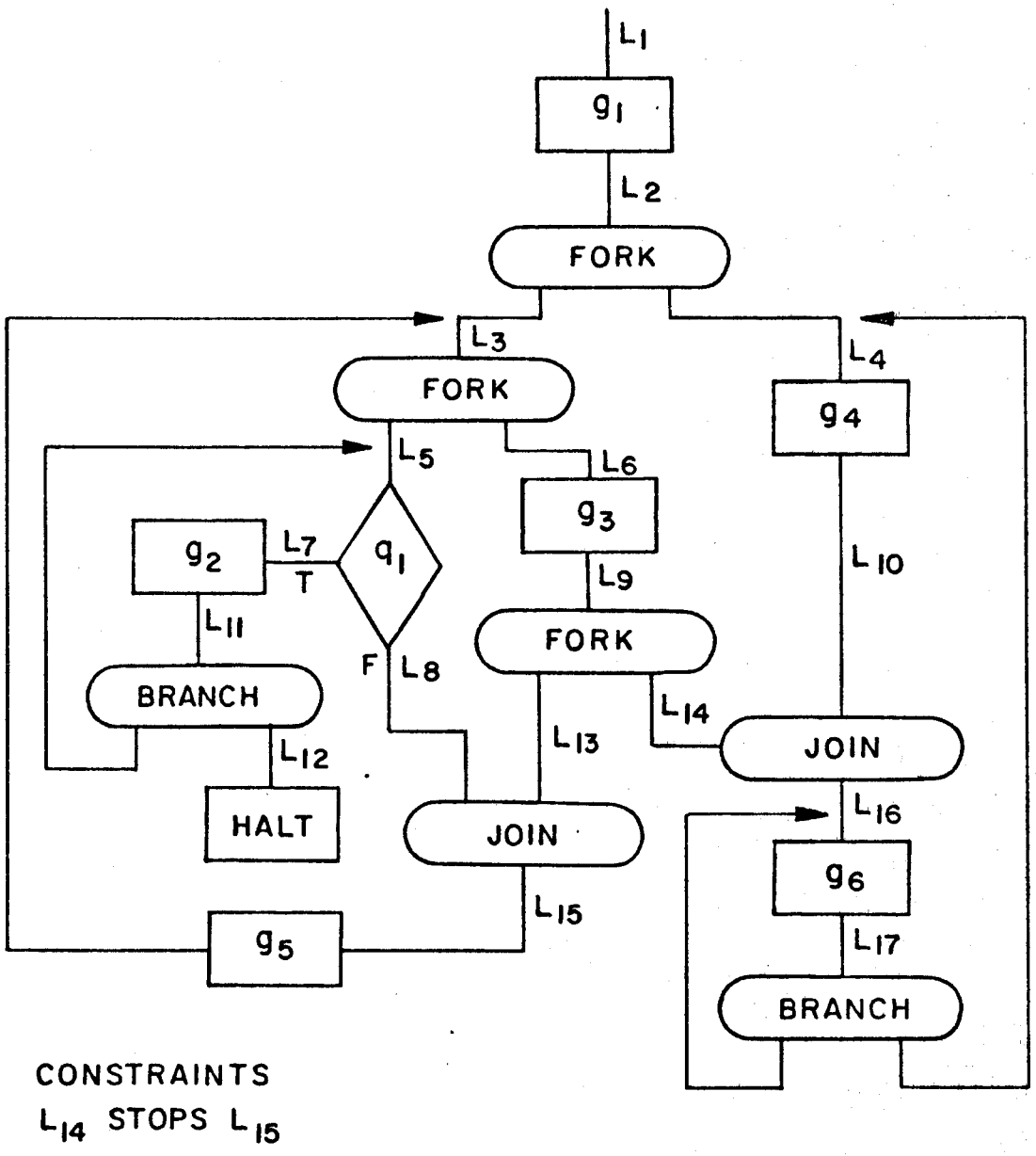

Figure 1 Example of a Parallel Program

We will always omit set braces around singleton sets when no confusion results, e.g. L_i $\underset{\sim}{\text{stops}}_P$ L_j means $\{L_i\}$ $\underset{\sim}{\text{stops}}_P$ L_j, and $\{L_1, L_2, \cdots, L_n\}$ $\underset{\sim}{\text{stop}}_P$ L_j means $\{\{L_1\}, \{L_2\}, \cdots, \{L_n\}\}$ $\underset{\sim}{\text{stop}}_P$ L_j, i.e. $\{L_i\}$ $\underset{\sim}{\text{stops}}_P$ L_j for $1 \leq i \leq n$.

We also define a converse of $\underset{\sim}{\text{stops}}_P$ which we call $\underset{\sim}{\text{allows}}_P$: for $c \in C_P$, $\ell \in L_P$, c $\underset{\sim}{\text{allows}}_P$ ℓ if, for all $c' \subseteq c$, c' $\underset{\sim}{\text{stops}}_P$ ℓ is false. Extending $\underset{\sim}{\text{allows}}_P$ to $C_P \times C_P$: c_1 $\underset{\sim}{\text{allows}}_P$ c_2 if, for all $\ell \in c_2$, c_1 $\underset{\sim}{\text{allows}}_P$ ℓ.

Semantics of Parallel Programs

To describe an execution of a program P we need not only the memory state at each instant, but also the control state. The control state represents the statements reached in the program; they are about to be executed at the instant in question.

Accordingly we let $Z_P = M_P \times C_P$ be the set of states.

We may describe an execution of P in the following intuitive way. We think of a control state as a set of markers on the corresponding statements. We start off with a state consisting of a single marker on the initial statement s_1 (i.e. the initial control state is $\{L_1\}$), and with some initial memory state m_o. At each step in the computation, with state (m, c), we choose some marker $\ell \in c$ at random such that c $\underset{\sim}{\text{allows}}_P$ ℓ (and if ℓ marks a join, we require that all labels ℓ' of the join be marked, and c $\underset{\sim}{\text{allows}}_P$ ℓ'). If ℓ marks an operation, then that statement is executed, m changes to $g_i(m)$ for some basic operation g_i and the marker is moved to the

next statement, changing c. In all other cases m is left unchanged, but c is updated. If ℓ marks a test, then the marker is moved to the appropriate next statement, depending on the basic test applied to m. If ℓ marks a fork, then the marker splits into several markers which are moved to the statements referred to in the fork. If ℓ marks a join, all the markers on the join are fused into one marker which moves to the statement referred to in the join. If ℓ marks a branch, then the marker is moved to one of the statements referred to in the branch, chosen at random. If ℓ marks a halt, then the marker is removed from c.

The process is repeated either indefinitely or until no marker may move, or no markers are left. Since control states are defined as subsets of L_P we will consider a program illegal if it allows any statement ever to get two markers. Syntactic restrictions could be imposed to ensure this, the details of which we shall not be concerned with here. (In fact it would not be difficult to remove this restriction and allow control states to be multisets instead of sets of labels.)

Example: A little study will show that in the example program of Figure 1, the big loops on the left and right are synchronized. The constraint ensures that the program is legal.

The formal definition of the semantics of P will be in two stages. Firstly we will define a next-state relation $\underset{\sim}{s}$ for every statement s in S_P, and then we will give a next-state relation $\underset{\sim}{S_P}$ for the whole program. To do this we first introduce the following important notation:

Control state decomposition notation

for $A,c \in C_P$

$A + c = A \cup c$ if A and c are disjoint, undefined otherwise.

We omit set braces on singleton sets:

$A + L_i$ means $A + \{L_i\}$.

Note that $c' = A + c$ means that c' is the disjoint union of A and c.

The relation $\underset{\sim}{s}$

for $(m, c),(m',c') \in Z_P$, $(m, c) \underset{\sim}{s} (m', c')$ is defined as follows:

i) operation: $(m, c)\underset{\sim}{L_i : \text{do } g_j \text{ then go to } L_k}(m', c')$ if and only if $c \underset{\sim}{\text{allows}_P} L_i$ &

$\exists A \in C_P[c = A + L_i \;\&\; m' = g_j(m) \;\&\; c' = A + L_k]$

ii) test: $(m, c)\underset{\sim}{L_i : \text{if } q_j \text{ then go to } L_k \text{ else go to } L_h}(m', c')$ if and only if $c \underset{\sim}{\text{allows}_P} L_i$ &

$\exists A \in C_P[c = A + L_i \;\&\; m = m' \;\&\; [\text{IF } q_j(m) \text{ THEN } c' = A + L_k \text{ ELSE } c' = A + L_h]]$*

iii) fork: $(m, c)\underset{\sim}{L_i : \text{go to } L_{\alpha_1}, L_{\alpha_2}, \cdots, L_{\alpha_j}}(m', c')$ if and only if $c \underset{\sim}{\text{allows}_P} L_i$ &

$\exists A \in C_P[c = A + L_i \;\&\; m = m' \;\&\; c' = A + \{L_{\alpha_1}, L_{\alpha_2}, \cdots, L_{\alpha_j}\}]$

---

* IF A THEN B ELSE $C \equiv (A \;\&\; B) \vee (\neg A \;\&\; C) \equiv (A \Rightarrow B) \;\&\; (\neg A \Rightarrow C)$

iv) join: $(m, c)\underset{\sim}{L_{\beta_1}, L_{\beta_2}, \cdots, L_{\beta_i} : \text{go to } L_j}(m', c')$ if and only if $c \underset{\sim}{\text{allows}_P} \{L_{\beta_1}, L_{\beta_2}, \cdots, L_{\beta_i}\}$ &

$\exists A \in C_P[c = A + \{L_{\beta_1}, L_{\beta_2}, \cdots, L_{\beta_i}\} \;\&\; m = m' \;\&\; c' = A + L_j]$

v) branch: $(m, c)\underset{\sim}{L_i : \text{go to one of } L_{\alpha_1}, L_{\alpha_2}, \cdots, L_{\alpha_j}}(m', c')$ if and only if $c \underset{\sim}{\text{allows}_P} L_i$ &

$\exists A \in C_P[c = A + L_i \;\&\; m = m' \;\&\; (c' = A + L_{\alpha_1} \vee c' = A + L_{\alpha_2} \vee \cdots \vee c' = A + L_{\alpha_j})]$

vi) halt: $(m, c)\underset{\sim}{L_i : \text{HALT}}(m', c')$ if and only if $c \underset{\sim}{\text{allows}_P} L_i$ &

$\exists A \in C_P[c = A + L_i \;\&\; m = m' \;\&\; c' = A]$

If P is a legal program, the relation $\underset{\sim}{s}$ clearly relates the states before and after execution of s in the way expected from the previous intuitive description. (If execution of s from some state z would result in duplicate markers then the decomposition notation would not be defined, and $z \underset{\sim}{s} z'$ is false, for all states z' .)

The relation $\underset{\sim}{S_P}$

for $z_1, z_2 \in Z_P$, $z_1 \underset{\sim}{S_P} z_2$ if and only if $z_1 \underset{\sim}{s} z_2$ for some $s \in S_P$. $\underset{\sim}{S_P}$ relates possible successive states in computations of P .

Computations of P

For $m_o \in M_P$, a computation of $P(m_o)$ is a sequence of

states $z_1, z_2, z_3, \cdots$ where $z_1 = (m_o, L_1)$ and $z_i \underset{\sim}{S}_P z_{i+1}$ for $i = 1, 2, 3, \cdots$. The sequence terminates, if at all, with a state z_n for which $z_n \underset{\sim}{S}_P z$ is false for all $z \in Z_P$. If the control state in z_n is the empty set then we have normal termination, otherwise we have complete deadlock.

COMMENTS ON THE PARALLEL PROGRAMMING MODEL

The model was designed to be simple, yet at the same time general enough to describe the more specialised constructs that are found in practical parallel programs. In particular the constraints feature is general enough to describe the effects of critical sections, synchronization, coroutines etc. The language is not intended to be a practical programming language; it really is a model of parallel programming, a language for describing parallel programs. For any program that can be described this way we will be able to apply the generalization of the assertion method, to be given in the next section.

One feature of practical parallel programs that is absent in our model is the 'finite delay property'. This property would ensure that, in all computations of a program, every marker which is not permanently stopped is eventually moved.

Besides being very difficult to formalize, the finite delay property is peculiar in that it is irrelevant as far the assertion

method is concerned. All it does is exclude certain infinite computations - it does not prevent any, otherwise possible, situations from occurring. It affects properties such as termination and equivalence, but not 'partial correctness'. For this reason it has not been included in the model. We will comment further on this subject in the observations at the end of the paper.

THE ASSERTION METHOD FOR PARALLEL PROGRAMS

In the usual assertion method, applied to simple flowcharts, we attach 'assertions', relations on memory states, to edges in the flowchart. In simple flowcharts we can consider control states to be single labels, and labels are attached to edges as we did in Fig. 1. Therefore we can think of this attaching of assertions to edges as associating the assertions with control states. If all the edges have associated assertions we can think of all the attached assertions together as specifying a single relation on states; state (m, ℓ) satisfies the relation if and only if m satisfies the assertion attached to ℓ. This is the way we think of associating assertions with parallel programs, and we call the relation on states an assignment.

If Q is an assignment for P, a relation on Z_P, we denote by Q^I the relation on M_P such that $Q^I(m) \iff Q(m, \{L_1\})$. Q^I is called the initial assertion of the assignment; intuitively it

is the assertion attached to the start of the program. An assignment Q will be said to be valid if all states produced in computations of P, for initial values satisfying the initial assertion Q^I, satisfy Q.

Valid assignments can express significant facts about programs. We shall develop a verification condition W_P, a relation on assignments, such that any assignment Q satisfying W_P is a valid assignment for P. If some desired property of P can be expressed as a valid assignment, we can prove that P has the property by showing that the assignment satisfies W_P.

Definitions

Let $\underset{\sim}{\text{yields}}_P$ be the reflexive transitive closure of $\underset{\sim}{S}_P$. For $\Phi \subseteq M_P$ (think of Φ as an initial assertion), we define a relation on states $\Phi\text{-}\underset{\sim}{\text{produces}}_P$ as follows: for $z \in Z_P$

$\Phi\text{-}\underset{\sim}{\text{produces}}_P$ (z) if and only if $(m, \{L_1\})$ $\underset{\sim}{\text{yields}}_P$ z

for some (initial) memory state m such that $\Phi(m)$.

Note that $\Phi\text{-}\underset{\sim}{\text{produces}}_P{}^I = \Phi$, since $\underset{\sim}{\text{yields}}_P$ is reflexive.

A valid assignment for P is a relation Q on states such that $Q^I\text{-}\underset{\sim}{\text{produces}}_P \subseteq Q$,

i.e. for all $z \in Z_P$, $Q^I\text{-}\underset{\sim}{\text{produces}}_P(z) \Rightarrow Q(z)$.

A Verification Condition for P

Since $\underset{\sim}{\text{yields}}_P$ contains the transitive closure of $\underset{\sim}{S}_P$,

for any $\Phi \subseteq M_P$, the relation Φ-produces$_P$ has the following property:

Proposition

for all $z_1, z_2 \in Z_P$, if Φ-produces$_P(z_1)$ and $z_1 \underset{\sim}{S}_P z_2$ then Φ-produces$_P(z_2)$.

We can derive from this property a condition $W_P(Q)$ on an arbitrary relation on states, Q :

$$W_P(Q) \equiv \forall z_1, z_2 \in Z_P \quad \text{if} \quad Q^I\text{-produces}_P(z_1) \quad \text{and}$$
$$Q(z_1) \quad \text{and} \quad z_1 \underset{\sim}{S}_P z_2 \quad \text{then} \quad Q(z_2) .$$

Note that for $Q \equiv \Phi$-produces$_P$, we have that $W_P(Q)$, using the Proposition and the fact that in this case $Q^I = \Phi$. In general the converse is not true but we have:

Verification Condition Theorem

For any assignment Q , if $W_P(Q)$ then Q^I-produces$_P \subseteq Q$, i.e. Q is a valid assignment for P .

Proof. Assume Q^I-produces$_P \not\subseteq Q$. Then there exists a state z such that Q^I-produces$_P(z)$, but $Q(z)$ is false. By the definition of Q^I-produces$_P$ and of yields$_P$ there exists a finite sequence of states $z_1, z_2, \ldots, z_n$ such that $z_1 = (m, \{L_1\})$ for some m such that $Q^I(m)$, and $z_i \underset{\sim}{S}_P z_{i+1}$ for $i = 1, 2, \ldots, n-1$, and $z_n = z$. Let z_j be the last state in this sequence for which $Q(z_j)$; such a state exists since $Q(z_1)$. Now $j < n$ since $Q(z_n)$ is false, so z_{j+1} exists.

Note that $Q^{I}\text{-produces}_P(z_j)$, and $Q(z_j)$ and $z_j \underset{\sim}{S}_P z_{j+1}$. Since $W_P(Q)$, we get $Q(z_{j+1})$, a contradiction.

□

The condition W_P can be stated in a more concise and convenient form by first defining a relation $\{s\}Q$ on states, for each statement $s \in S_P$ and assignment Q . We shall write $z\{s\}Q$ instead of $\{s\}Q(z)$, so that our notation deliberately resembles similar notation of Hoare [5]; $z\{s\}Q$ will be true if executing s from state z (when possible) gives a state satisfying Q .

The relation $\{s\}Q$

For $z = (m, c) \in Z_P$, $z\{s\}Q$ is defined as follows:

i) operation: $(m, c)\{L_i : \text{do } g_j \text{ then goto } L_k\}Q$ if and only if

$$\forall A \in C_P[c = A + L_i \ \& \ c \ \underset{\sim}{\text{allows}}_P \ L_i \Rightarrow Q(g_j(m), A+L_k)]$$

ii) test: $(m, c)\{L_i : \text{if } q_j \text{ then goto } L_k \text{ else goto } L_h\}Q$ if and only if

$$\forall A \in C_P[c = A + L_i \ \& \ c \ \underset{\sim}{\text{allows}}_P \ L_i \Rightarrow \text{IF } q_j(m) \text{ THEN } Q(m, A+L_k) \text{ ELSE } Q(m, A+L_h)]$$

iii) fork: $(m, c)\{L_i : \text{goto } L_{\alpha_1}, L_{\alpha_2}, \ldots, L_{\alpha_j}\}Q$ if and only if

$$\forall A \in C_P[c = A + L_i \ \& \ c \ \underset{\sim}{\text{allows}}_P \ L_i \Rightarrow Q(m, A + \{L_{\alpha_1}, L_{\alpha_2}, \ldots, L_{\alpha_j}\})]$$

iv) join: $(m, c)\{L_{\beta_1}, L_{\beta_2}, \ldots, L_{\beta_i} : \text{goto } L_j\}Q$ if and only if

$$\forall A \in C_P[c = A + \{L_{\beta_1}, L_{\beta_2}, \ldots, L_{\beta_i}\} \ \& \ c \ \underset{\sim}{\text{allows}}_P \ \{L_{\beta_1}, L_{\beta_2}, \ldots, L_{\beta_i}\} \Rightarrow Q(m, A+L_j)$$

v) branch: $(m, c)\{L_i$: goto one of $L_{\alpha_1}, L_{\alpha_2}, \ldots, L_{\alpha_j}\}Q$ if and only if

$$\forall A \in C_P[c = A + L_i \;\&\; c \underset{\sim}{\text{allows}}_P L_i \Rightarrow Q(m, A+L_{\alpha_1}) \;\&\; Q(m, A+L_{\alpha_2}) \;\&\; \ldots \;\&\; Q(m, A+L_{\alpha_j})]$$

vi) halt: $(m, c)\{L_i$: HALT$\}Q$ if and only if

$$\forall A \in C_P[c = A + L_i \;\&\; c \underset{\sim}{\text{allows}}_P L_i \Rightarrow Q(m, A)]$$

It follows immediately from the definitions that for $z_1 \in Z_P$

$$[\forall s \in S_P, z_1\{s\}Q]$$

if and only if $[\forall s \in S_P, \forall z_2 \in Z_P, z_1 \underset{\sim}{S} z_2 \Rightarrow Q(z_2)]$

if and only if $[\forall z_2 \in Z_P, z_1 \underset{\sim}{S}_P z_2 \Rightarrow Q(z_2)]$.

We can thus state W_P equivalently as follows:

Revised Definition of W_P

$$W_P(Q) \equiv \forall z \in Z_P\ , [Q \underset{\sim}{\text{-produces}}_P^I(z) \;\&\; Q(z)] \Rightarrow \forall s \in S_P, z\{s\}Q$$

In this form we shall call W_P the verification condition for P .

Using the Assertion Method

To prove that an assignment R for program P is valid one must check that $W_P(R)$ is true. This involves checking that for all states z such that $R^I\text{-produces}_P(z)$, and all statements $s \in S_P$,

$$R(z) \Rightarrow z\{s\}R \ .$$

This is not as formidable a task as it appears for the following reasons.

Firstly, in practice one would usually check the above condition for <u>all</u> states z, not just those for which $R^I\text{-produces}_P(z)$. This simplifies matters since we don't know exactly which states satisfy $R^I\text{-produces}_P$ (if we did then we could check if R were valid directly). In fact, without loss of generality, we could have left the term $R^I\text{-produces}_P$ out of the verification condition W_P entirely.

However, if the impossibility of reaching certain states is crucial for certain properties of a program to hold, then this impossibility of states must come into the proof of the properties somewhere. One way is to explicitly incorporate the impossibility into the assignment we wish to prove valid, and check the above condition for all states. Alternatively, if we can easily establish separately, from considerations of constraints etc., that the states in question are impossible, then we need not add the impossibility into the assignment; we simply don't bother to check the above condition for the impossible states. This second method makes proofs cleaner.

Summarising, the term $R^I\text{-produces}_P$ in the verification

condition is just there to indicate that we don't need to check impossible states.

The second observation is that in practice valid assignments tend to be surprisingly uniform. If a program has n statements then there are 2^n control states (though some may be impossible to achieve). It might appear that an assignment would then be essentially the union of 2^n different 'assertions', one per control state. (These are the assertions we would need if we translated the program to a non-deterministic program as in Ashcroft and Manna [2].) However, in practice no one could write parallel programs if he had to think of every possible control state individually. The program must be designed in such a way that it is possible to reason along the lines "if we are at this statement then such and such is true, no matter where else we are in the program". An assignment then becomes more like the union of n assertions.

The final observation is related to the previous one. Note that in checking

$$R(z) \quad => \quad z\{s\}R$$

we need only consider those states z for which R is already true and only those statements whose execution can make R become false. Taken together with the relatively uniform nature of R in practice, this results in a drastic reduction in the number of cases we need to consider.

The most convincing demonstration of these points is probably by an example.

EXAMPLE: AN ELEMENTARY AIRLINE RESERVATION SYSTEM

We shall consider a program P which simulates a simple airline reservation system for one flight with up to K passengers. Orders to book and unbook customers would normally be received by the system from remote terminals in travel agents' offices. We do not intend to describe this aspect of the system, and instead will simply simulate the 'kernel' of the system where routines for booking and unbooking customers are running in parallel, and are called in random fashion.

There is no subroutine feature in our language, but we will simulate the effect by having different copies of the routines for each customer, corresponding to different activations of the same subroutines. These subroutines are simply those for booking and unbooking a customer, and we will assume that each customer calls the subroutines repeatedly, in random order, with arbitrary delays at any time (since the language makes no assumptions about the running rates of separate computations). The copies of the subroutines for all the customers are run in parallel, together with a service routine to handle the waiting-list, transferring customers from the waiting-list to the flight-list when other customers cancel their bookings.

For convenience we shall number the customers 1,2,3,... and the outline of the system is then as shown in Figure 2.

The program maintains two lists of integers, in variables L and W , representing the flight-list and the waiting-list respectively. Before giving details of the subprograms, we will define the basic operations on lists that we shall use.

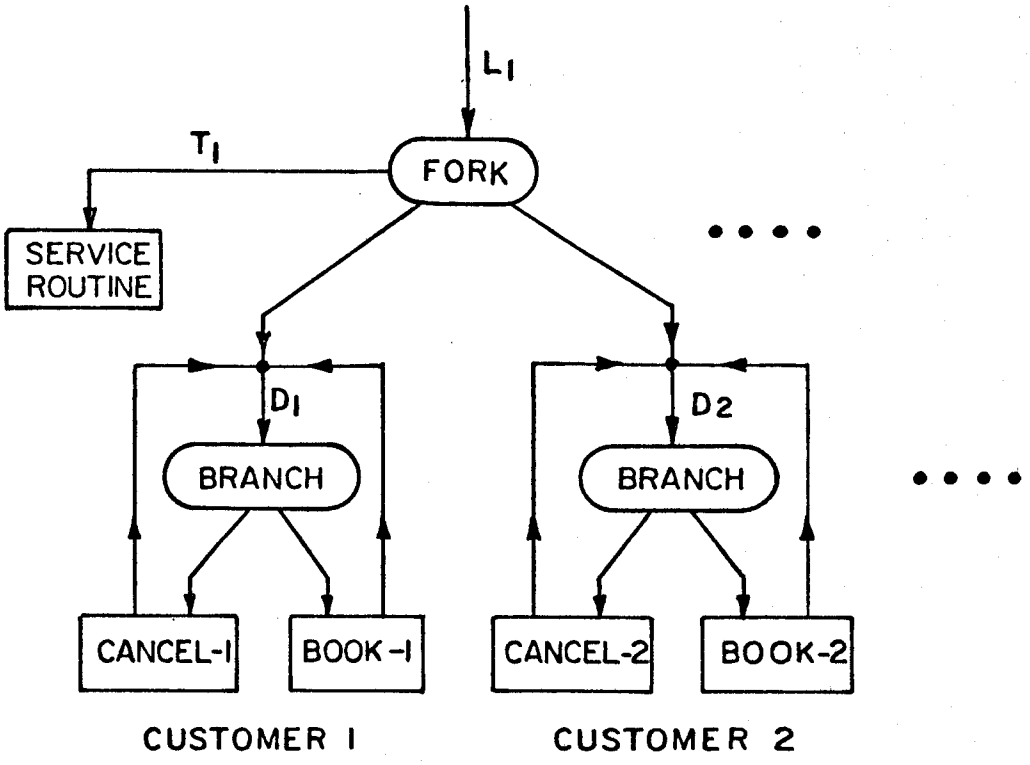

Figure 2 Outline of Airline Reservation System Simulation

Let J be the set of non-negative integers. For $x \in J^*$ and $j \in J$, $\circ$ denotes concatenation, Λ denotes the empty list and

i) $j \in x$ if $x = \alpha \circ j \circ \beta$ for some $\alpha, \beta \in J^*$

ii) $add(j, x) = x \circ j$

iii) $remove(j, x)$ is defined if $j \in x$ and then

$remove(j, x) = \alpha \circ \beta$ where $x = \alpha \circ j \circ \beta$ and $j \notin \alpha$

iv) $top(x)$ is defined if $x \neq \Lambda$, and then $top(x) = j$ where $x = j \circ \alpha$, $j \in J$.

v) $pop(x)$ is defined if $x \neq \Lambda$, and $pop(x) = remove(top(x), x)$

The program also associates with the i-th customer a variable M_i which will hold an integer representing the last message sent to the customer. The message codes should be read as follows

0	'not listed'
1	'booking cancelled'
2	'cancelled from waiting list'
3	'wait-listed'
4	'already on waiting list'
5	'booked'
6	'already booked'
7	'transferred from waiting to booked'

The copies of the booking and cancelling routines for customer-i are shown in Figures 3 and 4. The service routine is shown in Figure 5. The constraints are as follows (for all i, j):

i) $B_{i_7} \underset{\sim\sim}{stops}_P B_{j_5}$ $(i \neq j)$

ii) $B_{i_6} \underset{\sim\sim}{stops}_P T_2$, $T_3 \underset{\sim\sim}{stops}_P B_{i_5}$

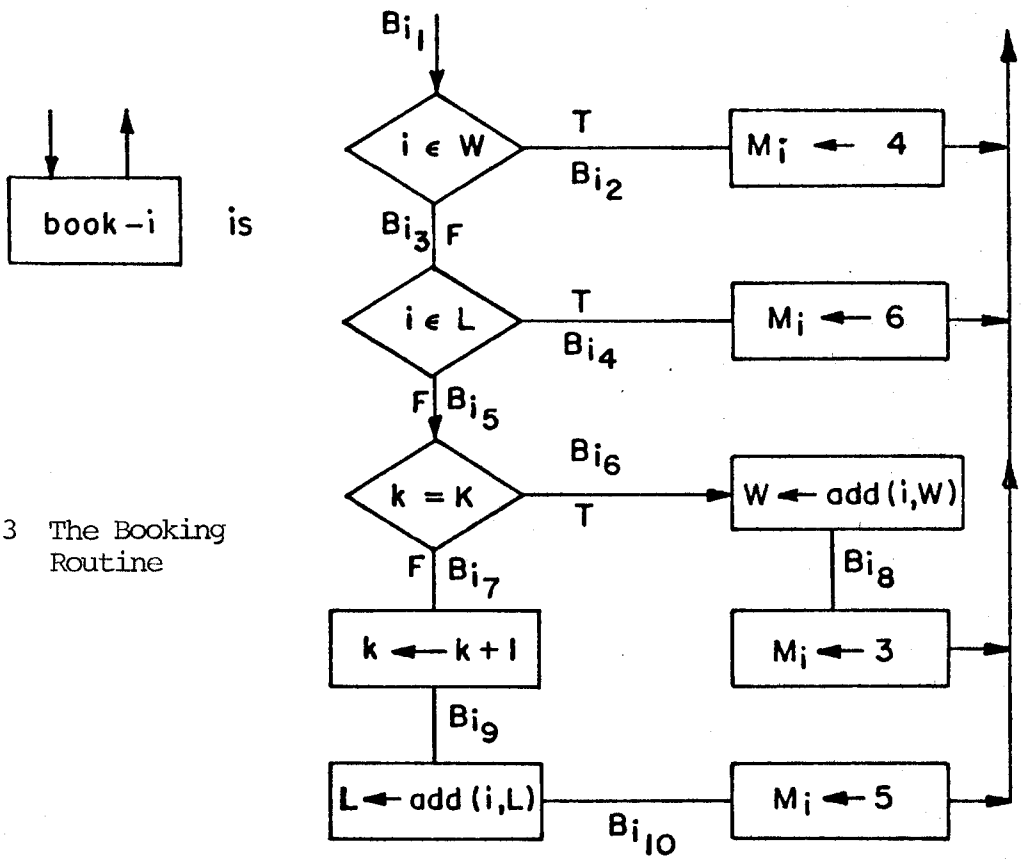

Figure 3 The Booking Routine

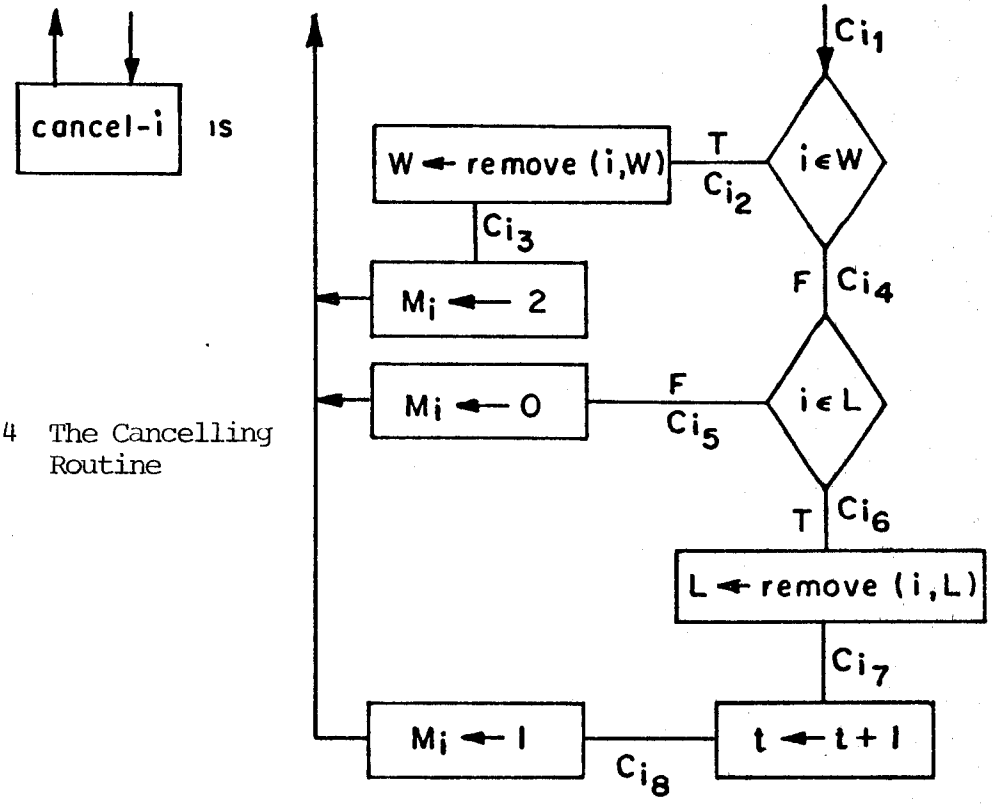

Figure 4 The Cancelling Routine

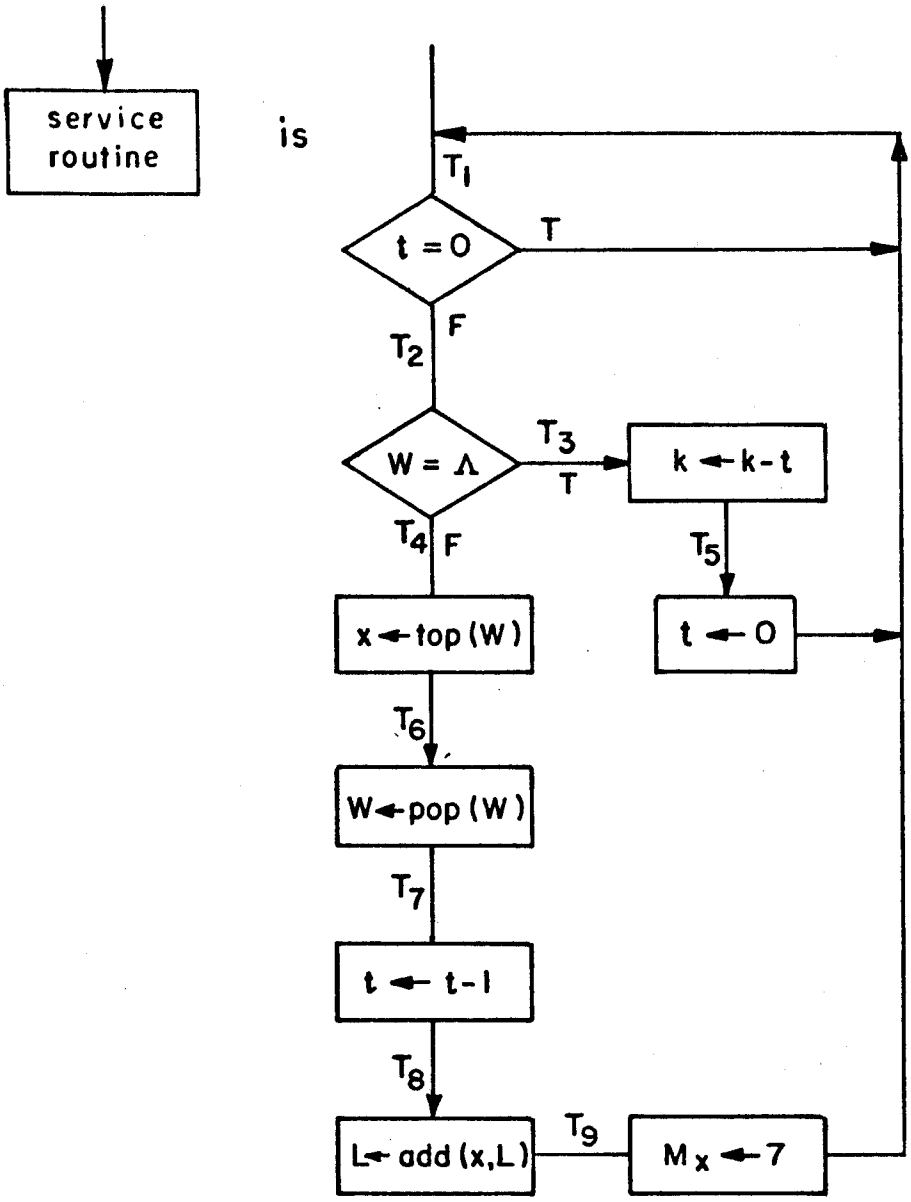

Figure 5 The Service Routine

iii) $C_{i_2} \underset{\sim}{\text{stops}}_P T_2$, $\{T_4, T_6\} \underset{\sim}{\text{stop}}_P C_{i_1}$

iv) $\{B_{i_2}, B_{i_8}\} \underset{\sim}{\text{stop}}_P T_9$, $T_9 \underset{\sim}{\text{stops}}_P \{B_{i_4}, C_{i_8}\}$

v) $\{T_7, T_8\} \underset{\sim}{\text{stop}}_P \{B_{i_3}, C_{i_4}\}$

vi) $T_5 \underset{\sim}{\text{stops}}_P C_{i_7}$

The routines are straightforward except for use of the variable t . This is used to indicate to the service routine how many customers should be transferred from the waiting list to the flight list. (The service routine could just keep topping up the flight list when seats became available, but then it would be possible for a customer booking at the appropriate instant to circumvent the waiting list and get booked directly.)

The constraints ensure that the program works correctly. They will all be used in proofs of valid assignments. Constraints i), ii) and iii) are used indirectly. They effectively set up 'critical sections' which ensure that no possible control states can contain

a) $\{B_{i_7}, B_{j_7}\}$ $(i \neq j)$,

b) $\{B_{i_6}, T_3\}$ or

c) $\{C_{i_2}, T_4\}$ or $\{C_{i_2}, T_6\}$

This is easily seen by considering the possible previous control states. For example, let $A + \{B_{i_6}, T_3\}$ be the first control state of type b) in some computation. The previous control state must have been $A + \{B_{i_5}, T_3\}$ or $A + \{B_{i_6}, T_2\}$, but from both of these

the desired transitions are not allowed. Hence $A + \{B_{i_6}, T_3\}$ does not occur.

Constraints iv) make sure that messages from the customer routines and the service routines don't get 'crossed'. Constraints v) make sure that a customer is properly transferred from the waiting list to the flight list before the customer can re-book or cancel. The last constraint makes sure the system can't 'forget' about cancelled seats.

The constraints may appear to be unnecessarily complicated. In a practical program, critical sections would be made much larger, and the pattern of constraints would be simpler. However the number of constraints would be larger, and thus the parallelism, or freedom of action, would be reduced. Of course, this would also make the program easier to understand. However we prefer here to illustrate the usefulness of the assertion method for really complex programs.

The Verification Condition for the Airline Reservation System

To construct the verification condition W_P we need to specify the basic operations and tests of P, or at least find suitable notation for them. The memory states of P will be vectors of values for the variables L, W, k, x, t and $M_1, M_2, \ldots$. The assignment statements in P each change only one of these values. We shall adopt the notation that the basic operation (mapping M_P into M_P) associated with each assignment statement a will be denoted by $[a]$. This notation

is concise and expresses just the relevant information: what has been changed. For example if $m = \langle \alpha, \beta, \gamma, \delta, \varepsilon, \psi_1, \psi_2, \ldots \rangle$ (values for L, W, k, x, t and $M_1, M_2, \ldots$ respectively) then

$$[W \leftarrow \text{remove } (i, W)](m) = \langle \alpha, \text{remove } (i, \beta), \gamma, \delta, \varepsilon, \psi_1, \psi_2, \ldots \rangle \ .$$

We shall also use this notation for basic tests, e.g. for the same m as above $[k = K](m)$ is true if and only if $\gamma = K$.

For readability we shall specify W_P using a relation $\underset{\sim}{at}$, representing an assignment, which we shall write infix, i.e. for $z = (m, c) \in Z_P$, we write $m \ \underset{\sim}{at} \ c$ instead of $\underset{\sim}{at}(m, c)$.

We can naturally partition the statements in P into

a) The outer fork statement

b) The separate subprograms that run in parallel - the routines for each customer, and the service routine.

We will partition W_P accordingly:

$$W_P(\underset{\sim}{at}) \equiv W_{fork}(\underset{\sim}{at}) \ \& \ W_{service}(\underset{\sim}{at}) \ \& \ W_{P_1}(\underset{\sim}{at}) \ \& \ W_{P_2}(\underset{\sim}{at}) \ \& \ \ldots$$

These components are specified as follows (we write conjuncts on separate lines and label them for the statement in question):

$W_{fork}(\underset{\sim}{at}) \equiv \forall z = (m, c) \in Z_P \ , \ \forall A \in C_P \ , \text{ if } m \ \underset{\sim}{at} \ c \text{ and } \underset{\sim}{at}\text{-}\overset{I}{\text{produces}}_P(z)$

then

L_1: $c = A + L_1 \Rightarrow m \ \underset{\sim}{at} \ A + \{T_1, D_1, D_2, \ldots\}$.

(book-i and cancel-i)

$W_{P_i}(\underset{\sim}{at}) \equiv \forall z = (m, c) \in Z_P$, $\forall A \in C_P$ if $m \underset{\sim}{at} c$ and

$\underset{\sim}{at^I\text{-produces}_P}(z)$ then

D_i: $c = A + D_i \Rightarrow m \underset{\sim}{at} A + B_{i_1}$ & $m \underset{\sim}{at} A + C_{i_1}$

B_{i_1}: $c = A + B_{i_1} \Rightarrow$ IF $[i \in W](m)$ THEN $m \underset{\sim}{at} A + B_{i_2}$ ELSE $m \underset{\sim}{at} A + B_{i_3}$

B_{i_2}: $c = A + B_{i_2} \Rightarrow [M_i \leftarrow 4](m) \underset{\sim}{at} A + D_i$

B_{i_3}: $c = A + B_{i_3}$ & $T_7 \notin c$ & $T_8 \notin c \Rightarrow$ IF $[i \in L](m)$ THEN $m \underset{\sim}{at} A + B_{i_4}$
ELSE $m \underset{\sim}{at} A + B_{i_5}$

B_{i_4}: $c = A + B_{i_4}$ & $T_9 \notin c \Rightarrow [M_i \leftarrow 6](m) \underset{\sim}{at} A + D_i$

B_{i_5}: $c = A + B_{i_5}$ & $(\forall i)B_{i_7} \notin c$ & $T_3 \notin c \Rightarrow$ IF $[k = K](m)$
THEN $m \underset{\sim}{at} A + B_{i_6}$ ELSE $m \underset{\sim}{at} A + B_{i_7}$

B_{i_6}: $c = A + B_{i_6} \Rightarrow [W \leftarrow add(i, W)](m) \underset{\sim}{at} A + B_{i_8}$

B_{i_7}: $c = A + B_{i_7} \Rightarrow [k \leftarrow k+1](m) \underset{\sim}{at} A + B_{i_9}$

B_{i_8}: $c = A + B_{i_8} \Rightarrow [M_i \leftarrow 3](m) \underset{\sim}{at} A + D_i$

B_{i_9}: $c = A + B_{i_9} \Rightarrow [L \leftarrow add(i, L)](m) \underset{\sim}{at} A + B_{i_{10}}$

$B_{i_{10}}$: $c = A + B_{i_{10}} \Rightarrow [M_i \leftarrow 5](m) \underset{\sim}{at} A + D_i$

C_{i_1}: $c = A + C_{i_1}$ & $T_4 \notin c$ & $T_6 \notin c \Rightarrow$ IF $[i \in W](m)$ THEN
$m \underset{\sim}{at} A + C_{i_2}$ ELSE $m \underset{\sim}{at} A + C_{i_4}$

C_{i_2}: $c = A + C_{i_2} \Rightarrow [W \leftarrow remove(i, W)](m) \underset{\sim}{at} A + C_{i_3}$

C_{i_3}: $c = A + C_{i_3} \Rightarrow [M_i \leftarrow 2](m) \underset{\sim}{at} A + D_i$

C_{i_4} : $c = A + C_{i_4}$ & $T_7 \notin c$ & $T_8 \notin c$ => IF $[i \in L](m)$ THEN $m \underset{\sim}{at} A + C_{i_6}$ ELSE $m \underset{\sim}{at} A + C_{i_5}$

C_{i_5} : $c = A + C_{i_5}$ => $[M_i \leftarrow 0](m) \underset{\sim}{at} A + D_i$

C_{i_6} : $c = A + C_{i_6}$ => $[L \leftarrow \text{remove}(i, L)](m) \underset{\sim}{at} A + C_{i_7}$

C_{i_7} : $c = A + C_{i_7}$ & $T_5 \notin c$ => $[t \leftarrow t+1](m) \underset{\sim}{at} A + C_{i_8}$

C_{i_8} : $c = A + C_{i_8}$ & $T_9 \notin c$ => $[M_i \leftarrow 1](m) \underset{\sim}{at} A + D_i$.

(service routine)

$W_{service}(\underset{\sim}{at}) \equiv \forall z = (m, c) \in Z_P$, $\forall A \in C_P$, if $m \underset{\sim}{at} c$ and $\underset{\sim}{at}^{I}\text{-produces}_P(z)$ then

T_1: $c = A + T_1$ => IF $[t = 0](m)$ THEN $m \underset{\sim}{at} c$ ELSE $m \underset{\sim}{at} A + T_2$

T_2: $c = A + T_2$ & $(\forall i)(C_{i_2} \notin c$ & $B_{i_6} \notin c)$ => IF $[W = \Lambda](m)$ THEN $m \underset{\sim}{at} A + T_3$ ELSE $m \underset{\sim}{at} A + T_4$

T_3: $c = A + T_3$ => $[k \leftarrow k-t](m) \underset{\sim}{at} A + T_5$

T_4: $c = A + T_4$ => $[x \leftarrow \text{top}(W)](m) \underset{\sim}{at} A + T_6$

T_5: $c = A + T_5$ => $[t \leftarrow 0](m) \underset{\sim}{at} A + T_1$

T_6: $c = A + T_6$ => $[W \leftarrow \text{pop}(W)](m) \underset{\sim}{at} A + T_7$

T_7: $c = A + T_7$ => $[t \leftarrow t-1](m) \underset{\sim}{at} A + T_8$

T_8: $c = A + T_8$ => $[L \leftarrow \text{add}(x, L)](m) \underset{\sim}{at} A + T_9$

T_9: $c = A + T_9$ & $(\forall i)(B_{i_2} \notin c$ & $B_{i_8} \notin c)$ => $[M_x \leftarrow 7](m) \underset{\sim}{at} A + T_1$.

Valid Predicates and Their Proofs

We will first show (A) that the flight never gets overbooked and that the flight is fully booked whenever there is a waiting list.

Then we will show (B) that for every customer, the message he last received always corresponds to his present booking status (except for those situations where his status has just changed and a new message is about to be sent).

A) Number of passengers booked

Intuitively, k is the number of seats reserved on the flight, for passengers already on the flight list or for passengers about to be transferred from the waiting list. Since K is the number of seats on the flight, we would require that $k \leq K$ at all times. Moreover if $k < K$ we would expect the waiting list to be empty. Variable t represents the number of passengers to be transferred from the waiting list, so we should expect that $k = |L| + t$, where $|L|$ denotes the length of flight-list L. Changes to L are accompanied by changes to k or t which will preserve this relationship. However, in those situations where L has been changed and k or t has not yet been changed, or vice versa, the above relationship will not hold. To correct for these situations we must keep account of them: let

$$\text{no.pending}(c) = \text{number of labels } T_8,\ B_{i_9} \text{ or } C_{i_7} \text{ (for some i) in } c .$$

Then, it is more true to say that

$$k = |L| + t + \text{no. pending}(c) \quad .$$

Unfortunately, there is one more situation we must take special account of; in the service routine, when the waiting list has become empty the number of reserved seats is reduced by t and t is then set to zero.

The above relationship will not hold in the situations occurring inbetween these actions.

The desired properties can now be expressed as an assignment:

$\underset{\sim}{R}_1$: for $m = \langle L, W, k, x, t, M_1, M_2, \ldots\rangle \in M_P$, $c \in C_P$, $m\underset{\sim}{R}_1 c$ if and only if

i) $(k < K \;\&\; W = \Lambda) \vee k = K$

ii) $t \geq 0$

iii) $k - |L| - \text{no. pending}(c) = \underline{\text{if}}\ T_5 \in c\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ t$

To show that $\underset{\sim}{R}_1$ is valid, we actually have to prove a stronger property; we need to know for example <u>when</u> k is less than K or W is empty:

$\underset{\sim}{R}_2$: for $m = \langle L, W, k, x, t, M_1, M_2, \ldots\rangle \in M_P$, $c \in C_P$, $m\underset{\sim}{R}_2 c$ if and only if

iv) $W = \Lambda$ when $T_3 \in c$

v) $k = K$ when $B_{i_6} \in c$, for some i

vi) $k < K$ when $B_{i_7} \in c$, for some i

vii) $t > 0$ when T_2, T_4, T_6 or $T_7 \in C$.

<u>Proposition</u> $\underset{\sim}{R}_3 = \underset{\sim}{R}_1 \cap \underset{\sim}{R}_2$ is a valid assignment for P .

<u>Proof</u> To check that $W_P(\underset{\sim}{R}_3)$ is true we need only check the transitions (from possible states) that can make $\underset{\sim}{R}_3$ false.

<u>$W_{\text{fork}}(\underset{\sim}{R}_3)$</u> : Executing statement L_1 cannot make $\underset{\sim}{R}_3$ false.

<u>$W_{P_i}(\underset{\sim}{R}_3)$</u> : The only statements whose execution can affect $\underset{\sim}{R}_3$ are checked below:

$$B_{i_5} : mR_3c \;\&\; c = A + B_{i_5} \;\&\; \ldots$$

$$\Rightarrow \text{IF } [k = K](m) \text{ THEN } mR_3\, A+B_{i_6}$$

$$\text{ELSE } mR_3\, A+B_{i_7}$$

Only v) and vi) are affected. $k = K$ is necessary for the transition to B_{i_6}, $k < K$ is necessary to get to B_{i_7}. Therefore v) and vi) are satisfied.

$$B_{i_6} : mR_3c \;\&\; c = A + B_{i_6} \Rightarrow [W \leftarrow \text{add}(i, W)](m)R_3\, A+B_{i_8}$$

Only i) and iv) affected. Before the transition, v) implies $k = K$, therefore i) is satisfied afterwards. No control state can contain $\{B_{i_6}, T_3\}$, so v) is satisfied after the transition.

$$B_{i_7} : mR_3c \;\&\; c = A + B_{i_7} \Rightarrow [k \leftarrow k+1](m)R_3\, A+B_{i_9}$$

Only i), iii), v) and vi) affected. Both k and no. pending(c) are increased by one, so iii) remains true. From vi), $k < K$ before the transition, so i) is still true after the transition. Also, since v) and $k < K$ are both true before the transition, the control state can not contain B_{j_6} for some j, so v) is true after the transition. Since no control state can contain $\{B_{i_7}, B_{j_7}\}$, vi) will be true after the transition.

$$B_{i_9} : mR_3c \;\&\; c = A + B_{i_9} \Rightarrow [L \leftarrow \text{add}(i, L)](m)R_3\, A+B_{i_{10}}$$

Only iii) is affected. $|L|$ is increased by one, but no.pending(c) is decreased by one, so iii) remains true.

C_{i_6} : mR_3c & $c = A + C_{i_6}$ $\Rightarrow$ $[L \leftarrow \text{remove}(i, L)](m)R_3$ $A+C_{i_7}$

Only iii) affected. After the transition, $|L|$ is decreased by one (if L was empty, the transition would not be completed), but no.pending(c) is increased by one, so iii) remains true.

C_{i_7} : mR_3c & $c = A + C_{i_7}$ & $T_5 \notin c$ $\Rightarrow$ $[t \leftarrow t+1](m)R_3$ $A+C_{i_8}$

Only ii) and iii) affected. ii) clearly remains true. After the transition, t is increased by one, but no.pending(c) is decreased by one. Since $T_5 \notin c$ for the transition to occur , iii) remains true.

$W_{service}(R_3)$: The following are the statements in the service routine that can affect R_3 :

T_1 : mR_3c & $c = A + T_1$ $\Rightarrow$ IF $[t = 0](m)$ THEN ...
ELSE mR_3 $A+T_2$

Only vii) affected. Since $t \geq 0$ by ii) and $t \neq 0$ is necessary for the transition, vii) is satisfied.

T_2 : mR_3c and $c = A + T_2$ & ... $\Rightarrow$ IF $[W = \Lambda](m)$
THEN mR_3A+T_3 ELSE mR_3A+T_4

Only iv) and vii) affected. vii) ensures that $t > 0$ before the transition, so vii) true afterwards. iv) is satisfied since $W = \Lambda$ is necessary for the transition to T_3 .

$$T_3 : m\underset{\sim}{R}_3 c \;\&\; c = A + T_3 \Rightarrow [k \leftarrow k-t]\,(m)\underset{\sim}{R}_3\, A+T_5$$

Only i), iii), v) and vi) affected. iv) implies $W = \Lambda$ before the transition, and $t \geq 0$ by ii), therefore i) remains true afterwards. k is decreased by t, but we move to T_5, so right hand side of iii) also decreases by t, and iii) remains true. k does not decrease since $t \geq 0$ by ii), so vi) must remain true. No control state can contain $\{B_{i_6}, T_3\}$, so v) is true after the transition.

$$T_4 : m\underset{\sim}{R}_3 c \;\&\; c = A + T_4 \Rightarrow [x \leftarrow top(W)]\,(m)\underset{\sim}{R}_3 A+T_6$$

Only vii) affected, but it must remain true, since vii) implies $t > 0$ before the transition.

$$T_5 : m\underset{\sim}{R}_3 c \;\&\; c = A + T_5 \Rightarrow [t \leftarrow 0]\,(m)\underset{\sim}{R}_3\, A+T_1$$

Only iii) affected, and it remains true because its right hand side was already zero.

$$T_6 : m\underset{\sim}{R}_3 c \;\&\; c = A + T_6 \Rightarrow [W \leftarrow pop(W)]\,(m)\underset{\sim}{R}_3\, A+T_7$$

Only vii) affected (shortening W cannot make $\underset{\sim}{R}_3$ false). From vii), $t > 0$ before the transition, so vii) is true afterwards.

$$T_7 : m\underset{\sim}{R}_3 c \;\&\; c = A + T_7 \Rightarrow [t \leftarrow t-1]\,(m)\underset{\sim}{R}_3\, A+T_8$$

Only ii) and iii) affected. By vii), $t > 0$ before the transition, ii) is true afterwards. iii) remains true because t is decreased by one, but no.pending(c) is increased by one.

$$T_8 : m R_3 c \ \& \ c = A + T_8 \Rightarrow [L \leftarrow add(x, L)](m) R_3 \ A+T_9$$

Only iii) affected, and it remains true because $|L|$ is increased by one, but no.pending(c) is decreased by one.

No transition can make R_3 false, so $W_P(R_3)$ is true and $R_1 \cap R_2$ is a valid assignment for P.

□

Thus, any computation of P that starts off with values of L, W, k and t satisfying R_1 will always have values satisfying R_1.

B) Correctness of messages

We prove that at positions D_i, B_{i_1} and C_{i_1} the last message M_i received by customer i corresponds to his booking status, unless he is in the process of being transferred by the service routine. At the same time we check that no double booking occurs.

We define i-transferring$_1$(c) to mean T_7 or $T_8 \in c$ & $x = i$; also i-transferring$_2$(c) means $T_9 \in c$ & $x = i$. Then i-transferring(c) means i-transferring$_1$(c) or i-transferring$_2$(c).

The assignment S_{i_1} We wish to prove valid is

S_{i_1} : For $m = \langle L, W, k, x, t, M_1, M_2, \ldots \rangle \in M_P$, $c \in C_P$,

$m S_{i_1} c$ if and only if

i) $M_i \leq 2 \Rightarrow i \notin W \ \& \ i \notin L$

$M_i \geq 5 \Rightarrow i \in L$

$2 < M_i < 5 \Rightarrow i \in W \vee$ i-transferring(c)

when D_i, B_{i_1} or $C_{i_1} \in c$.

ii) $i \in L \Rightarrow i \notin W \ \& \ i \notin$ remove(i, L)

iii) $i \in W \Rightarrow i \notin L \ \& \ i \notin$ remove(i, W)

To prove $\underset{\sim}{S}_{i_1}$ is valid we need to prove a stronger assignment $\underset{\sim}{S}_{i_3} = \underset{\sim}{S}_{i_1} \cap \underset{\sim}{S}_{i_2}$ where we have:

$\underset{\sim}{S}_{i_2}$: For $m = \langle L, W, k, x, t, M_1, M_2, \ldots \rangle \in M_P$, $c \in C_P$, $m \underset{\sim}{S}_{i_2} c$ if and only if

iv) $i \notin W \ \& \ i \notin L \ \& \ (2 < M_i < 5 \vee B_{i_2} \in c)$ when i-transferring$_1$(c)

v) $i \in L \vee C_{i_7}$ or $C_{i_8} \in c$ when i-transferring$_2$(c)

vi) $W \neq \Lambda$ when T_4 or $T_6 \in c$

vii) $x = top(W)$ when $T_6 \in c$

viii) $i \in W \vee$ i-transferring(c) when B_{i_2} or $B_{i_8} \in c$

ix) $i \notin W$ when B_{i_3} or $C_{i_4} \in c$

x) $i \in L$ when B_{i_4}, $B_{i_{10}}$ or $C_{i_6} \in c$

xi) $i \in W$ when $C_{i_2} \in c$

xii) $i \notin L \ \& \ i \notin W \ \& \ \neg\, i\text{-transferring}_1(c)$ when B_{i_5}, B_{i_6}, B_{i_7}, B_{i_9}, C_{i_3}, C_{i_5}, C_{i_7} or $C_{i_8} \in c$.

Proposition $S_{i_3} = S_{i_1} \cap S_{i_2}$ is a valid assignment for P.

Proof To check $W_P(S_{i_3})$ is true we need only check those transitions that can make S_{i_3} false. None of the transitions in book-j or cancel-j, for $j \neq i$, can do this, so we need only check W_{fork}, W_{P_i} and $W_{service}$.

$W_{fork}(S_{i_3})$: executing L_1 cannot make S_{i_3} false.

$W_{P_i}(S_{i_3})$: The only statements whose execution can make S_{i_3} false are checked below:

B_{i_1} : $mS_{i_3} c \ \& \ c = A + B_{i_1} \Rightarrow$ IF $[i \in W](m)$ THEN $mS_{i_3} A + B_{i_2}$ ELSE $mS_{i_3} A + B_{i_3}$

Only viii) & ix) affected, and both are satisfied by the conditions for the corresponding transitions.

B_{i_2} : $mS_{i_3} c \ \& \ c = A + B_{i_2} \Rightarrow [M_i \leftarrow 4](m) S_{i_3} A + D_i$

Only i) affected, and is satisfied because viii) is true before the transition.

B_{i_4} : $mS_{i_3} c \ \& \ c = A + B_{i_3} \ \& \ T_7 \notin c \ \& \ T_8 \notin c \Rightarrow$ IF $[i \in L](m)$ THEN $mS_{i_3} A + B_{i_4}$ ELSE $mS_{i_3} A + B_{i_5}$

Only x) and xii) affected. x) is satisfied because $i \in L$ necessary for the transition to B_{i_4}. From ix) and because $i \notin L$ and $\neg$ i-transferring$_1$(c) are necessary for the transition to B_{i_5}, xii) is satisfied afterwards.

B_{i_4} : $mS_{\sim i_3}\, c \;\&\; c = A + B_{i_4} \;\&\; \ldots \Rightarrow [M_i \leftarrow 6](m)S_{\sim i_3}\, A + D_i$.

Only i) and iv) affected. i) is satisfied because x) is true before the transition. Also x) and iv) being true before the transition imply $\neg$ i-transferring$_1$(c), so iv) is true afterwards.

B_{i_6} : $mS_{\sim i_3}\, c \;\&\; c = A + B_{i_6} \Rightarrow [W \leftarrow \text{add}(i, W)](m)S_{\sim i_3}\, A + B_{i_8}$

Only ii), iii), iv), vii) & viii) affected. xii) is true before the transition, so ii), iii) & iv) are satisfied afterwards. viii) is satisfied by the effect of the transition. From vii) being true before the transition, if $T_6 \in c$ then W was non-empty before the transition and so vii) remains true afterwards.

B_{i_8} : $mS_{\sim i_3}\, c \;\&\; c = A + B_{i_8} \Rightarrow [M_i \leftarrow 3](m)S_{\sim i_3}\, A + D_i$

Only i) affected, and is satisfied because viii) was true before the transition.

B_{i_9} : $mS_{\sim i_3}\, c \;\&\; c = A + B_{i_9} \Rightarrow [L \leftarrow \text{add}(i, L)](m)S_{\sim i_3}\, A + B_{i_{10}}$

Only ii), iii), iv) & x) affected. x) is satisfied by the effected of the transition. Because xii) true before

transition, iv) is satisfied, and so are ii) and iii).

$B_{i_{10}}$: $m\underset{\sim}{S}_{i_3}\ c \ \&\ c = A + B_{i_{10}} \Rightarrow [M_i \leftarrow 5](m)\underset{\sim}{S}_{i_3}\ A + D_i$

Only i) and iv) affected. $i \in L$ because x) is true before the transition, so i) is satisfied. Also $\neg$ i-transferring$_1$(c) , from iv), so iv) remains true.

C_{i_1} : $m\underset{\sim}{S}_{i_3}\ c \ \&\ c = A + C_{i_1} \ \&\ \ldots \Rightarrow$ IF $[i \in W](m)$

THEN $m\underset{\sim}{S}_{i_3}\ A + C_{i_2}$ ELSE $m\underset{\sim}{S}_{i_3}\ A + C_{i_4}$

Only ix) and xi) affected. Both are satisfied because of the conditions for the transitions.

C_{i_2} : $m\underset{\sim}{S}_{i_3}\ c \ \&\ c = A + C_{i_2} \Rightarrow [W \leftarrow \text{remove}(i, W)](m)\underset{\sim}{S}_{i_3}\ A + C_{i_3}$

Only vi), vii) & xii) affected. Since no control state can contain $\{C_{i_2}, T_4\}$ or $\{C_{i_2}, T_6\}$, vi) and vii) remain true. From xi), iii) & iv) we have $i \in W$ and hence $\neg$ i-transferring$_1$(c) and $i \notin L$ before the transition, and $i \notin W$ after the transition. Therefore xii) is satisfied.

C_{i_3} : $m\underset{\sim}{S}_{i_3}\ c \ \&\ A=A + C_{i_3} \Rightarrow [M_i \leftarrow 2](m)\underset{\sim}{S}_{i_3}\ A + D_i$

Only i) & iv) affected. Both are satisfied because xii) is true before the transition.

C_{i_4} : $m\underset{\sim}{S}_{i_3}\ c \ \&\ c = A + C_{i_4} \ \&\ T_7 \notin c \ \&\ T_8 \notin c \Rightarrow$ IF $[i \in L](m)$

THEN $m\underset{\sim}{S}_{i_3}\ A + C_{i_6}$ ELSE $m\underset{\sim}{S}_{i_3}\ A + C_{i_5}$

Only x) and xii) affected. x) is satisfied because $i \in L$ necessary for the transition to C_{i_6}. From ix) and because $i \notin L$ and $\neg$ i-transferring$_1$(c) are necessary for the transition to C_{i_5}, xii) is satisfied afterwards.

C_{i_5} : $mS_{\sim i_3}\ c \ \&\ c = A + C_{i_5} \Rightarrow [M_i \leftarrow 0](m)S_{\sim i_3}\ A + D_i$

Only i) and iv) affected. xii) implies $i \notin L$ and $\neg$ i-transferring$_1$(c), so i) and iv) are both satisfied.

C_{i_6} : $mS_{\sim i_3}\ c \ \&\ c = A + C_{i_6} \Rightarrow [L \leftarrow \text{remove}(i, L)](m)S_{\sim i_3}\ A + C_{i_7}$

Only v) and xii) affected. From x), ii) and iv) we have $i \in L$ and hence $\neg$ i-transferring$_1$(c) and $i \notin W$ before the transition, and $i \notin L$ afterwards. Therefore xii) is satisfied, Since we move to C_{i_7}, v) is also satisfied.

C_{i_8} : $mS_{\sim i_3}\ c \ \&\ c = A + C_{i_8} \ \&\ T_9 \notin c \Rightarrow [M_i \leftarrow 1](m)S_{\sim i_3}\ A + D_i$

Only i), iv) and v) affected. xii) implies $i \notin L$ and $\neg$ i-transferring$_1$(c), so i) and iv) are satisfied. Since the transition cannot take place if transferring$_2$(c), v) is also satisfied.

$W_{\text{service}}(S_{i_3})$: The statements that can make $S_{\sim i_3}$ false are checked below:

T_2 : $m\underset{\sim}{S}_{i_3} c$ & $c = A + T_2$ & ... $\Rightarrow$ IF $[W = \Lambda](m)$

THEN $m\underset{\sim}{S}_{i_3} A + T_3$ ELSE $m\underset{\sim}{S}_{i_3} A + T_4$

Only vi) affected, and is satisfied because $W \neq \Lambda$ necessary for the transition to T_4 .

T_4 : $m\underset{\sim}{S}_{i_3} c$ & $c = A + T_4 \Rightarrow [x \leftarrow top(W)](m)\underset{\sim}{S}_{i_3} A + T_6$

Only vii) is affected, and is satisfied by the effect of the transition. vi) remains true because W is not changed.

T_6 : $m\underset{\sim}{S}_{i_3} c$ & $c = A + T_6 \Rightarrow [W \leftarrow pop(W)](m)\underset{\sim}{S}_{i_3} A + T_7$

Only i), iv), viii), xi) and xii) affected. From vi) and vii) we know that $W = \Lambda$ and $x = top(W)$ before the transition. We will have i-transferring$_1$(c) after the transition if and only if $x = i = top(W)$ before the transition. If we don't get to i-transferring$_1$(c), then i), iv) and viii) all remain true (i is not removed from W). If we do remove i from W we get to i-transferring$_1$(c), and again i), iv) and viii) remain true. We can only make xii) become false if $i \notin W$ before the transition and i-transferring$_1$(c) after the transition, which is impossible. xi) is satisfied because no control state can contain $\{C_{i_2}, T_6\}$.

T_8 : $m\underset{\sim}{S}_{i_3} c$ & $c = A + T_8 \Rightarrow [L \leftarrow add(x, L)](m)\underset{\sim}{S}_{i_3} A + T_9$

Only i), ii), iii), v) and xii) affected. To affect i), ii), iii) and xii) we must add i to L , which will only

be the case if we have i-transferring$_1$(c) before the transition and i-transferring$_2$(c) (and $i \in L$) after the transition. v) and xii) are immediately satisfied, and ii) and iii) are satisfied from iv). Also iv) implies that i) is satisfied.

T_9 : $mS_{\sim i_3}\ c$ & $c = A + T_9$ & ... $\Rightarrow$ $[M_x \leftarrow 7](m)S_{\sim i_3}\ A + T_1$

Only i) affected. By v), if D_i, B_{i_1} or $C_{i_1} \in c$ after the transition then $i \in L$, so i) is satisfied.

No transition can make $S_{\sim i_3}$ false, so $W_P(S_{\sim i_3})$ is true and $S_{\sim i_1} \cap S_{\sim i_2}$ is a valid assignment for P.

□

Thus, any computation that starts off with values of L, W and M_i satisfying $S_{\sim i_1}$ will always have values satisfying $S_{\sim i_1}$. (Note that the validity of $S_{\sim i_3}$ also ensures that the program never 'hangs up' by trying to remove a customer from a list when he is not present on the list.)

OBSERVATIONS

1. Both these proofs were long, but it can be argued that they are as short as any convincing proofs could be. The behaviour of the program is really very complex, and the proofs follow quite closely the reasoning that must be made in any proof. The advantage of the method over more informal reasoning is clearly that it brings out

all the cases. In fact many of the constraints were found to be necessary while attempting these proofs. Although the program appeared to behave in the desired way, in fact it didn't in certain circumstances which were discovered by attempting the proofs. Of course, a reasonable program would have much simpler behaviour, produced by having larger 'critical sections'. We have deliberately considered this program to show that the method can handle great complexity.

2. For parallel programs, proofs of assertion can not be simplified by applying the 'one assertion per loop' rule for sequential programs. Between the execution of any two textually successive statements in a program there may occur abitrary amounts of computation, elsewhere in the program. Nevertheless, some simplifications in applying the method may well be found.

3. Despite these proofs, we can not claim that the program is 'correct'. Even if we do not consider the problem of what 'correctness' could mean, we see that there are some desirable properties of the program that we have not proved, and which seem to be beyond the scope of the assertion method. For example, we have not proved that the waiting list works correctly - that no one can 'jump the queue'. Any properties relating situations at different times can not be handled.

   A crucial requirement for the program might be - if ever customer i enters book-i eventually he will get some response from the system, i.e. he will get a new message M_i and exit from book-i.

We have shown that if he gets a message it correctly reflects his status on the various lists, and we can even check that he gets put on or taken off the lists in the appropriate situations. But we can not be sure that he will eventually get out of book-i.

It might appear that this is where we can resurrect the finite delay property. We can show there are no deadlocks in the program, so if we have the finite delay property, each of the subprograms should keep going at some finite rate, ensuring that we get out of book-i, for example. Unfortunately, even without deadlocks, it is possible for book-i to be permanently stopped (and so even the finite delay property cannot help us)! For example, assume customers i, j and k are all on the waiting list, and customer i tries to book again just as he is being transferred to the flight list by the service routine. We can end up with a control state containing $\{B_{i_4}, T_9\}$, and B_{i_4} is stopped by T_9 to ensure that the messages don't get 'crossed'. Now imagine that customer j tries to book again, and gets to B_{j_2}, and also customer k tries to book, reaching B_{k_2}. Now T_9 is stopped as well as B_{i_4}. By the finite delay property, eventually B_{j_2} and B_{k_2} must be executed, but if B_{j_2} goes first, say, there is nothing to prevent customer j trying to rebook before B_{k_2} is executed, stopping T_9 with B_{j_2} once more. B_{k_2} can now be executed, but we can get back to B_{k_2} before B_{j_2} moves. And so on. We see that it is <u>possible</u> for T_9 to be permanently stopped <u>by a continually changing pattern of constraints</u>. This situation is

not deadlock, and in fact is much more difficult to detect. We might coin the term 'livelock' to describe it.

So we see that the program in fact is not correct if we require every subprogram to keep going. Admittedly the circumstances under which livelock can occur in this program are rather esoteric, but they may not be in other programs. It is probably possible to construct parallel programs where, for example, a round-robin algorithm, to decide which statement to execute next, gives precisely the behaviour which results in livelock of some process.

REFERENCES

[1] E. A. Ashcroft and Z. Manna "Formalization of Properties of Parallel Programs", Machine Intelligence 6, Edinburgh University Press (1970).

[2] R. W. Floyd "Assigning Meaning to Programs", Proc. Symposia in Appl. Math. 19, Amer. Math. Soc. (1967).

[3] P. B. Hansen "A Comparison of Two Synchronizing Concepts", Acta Informatica 1, No. 3 (190-199) (1972).

[4] R. Karp and R. Miller "Parallel Program Schemata", J. Comput. System Sci. 3, 147-195.

[5] K. N. Levitt "The Application of Program-proving Techniques to the Verification of synchronization Processes", Proceedings Fall Joint Comp. conference, 1972, 33-47.

[6] Z. Manna "The Correctness of Programs", J. Comput. System Sci. 3, No. 2 (May 1969).

[7] Z. Manna and A. Pnueli "Formalization of Properties of Functional Programs", J. Assoc., Comput. March 17, No. 3 (July 1970).