LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Proving Packet Communications Architectures Correct

Computation Structures Group Memo 143
September 1976

Glen Seth Miranker

This research was supported in part by the National Science Foundation under grant DCR75-04060.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

The construction of formal models of programming languages has allowed people to make precise and rigorous statements about the semantics of modelled languages. In addition it has provided the means for presenting rigourous proofs of properties of programs such as equivalence and termination. In this paper we also wish to make a precise and rigourous statement, not about properties of a programming language, but about the correctness of a machine architecture. Informally, what we mean by this is simple. Namely, every machine executes an encoding of some language which we call the base language of the machine. We can regard programs in the base language as encodings of a class of (interpreted) program schemas Π. The question then arises as to whether this machine is equivalent to the class of schemas Π in the sense that every schema has an encoding on the machine and for every such interpreted schema that terminates does the machine "compute the right thing". This question becomes more interesting if one first proposes the schemas and then designs a machine which is supposed to correctly implement them. In this note we will give a definition of what it means (formally) for a machine architecture to be correct. The definition is particularly appropriate to a class of machines known as packet communications architectures [2]. We will give an example of a proof of a machine using this definition by proving that the elementary data flow processor (EDFP) [3] correctly implements an interesting sub-class of the well formed data flow schemas (WFDFS) [1]. It will be seen that the structure of the proof exploits the modular construction of these machines.

Let M be an autonomous non-deterministic finite state machine (ANDFSM). We can completely characterize M by a quadruple $<S_M, S_M^0, S_M^f, \Phi_M>$ where

S_m = set of states of M
$S_m^o \subseteq S$ are the initial or start states of M
$S_m^f \subseteq S$ are the final state of M
Φ_m is a binary relation on S_m

If $(s_1, s_2) \in \Phi_m$

then we say that M can undergo a transition from s_1 to s_2 alternately we say the that (s_1, s_2) is a legal transiton.

For convenience we define the function δ_m: $S \to 2^S$ where

$\delta_m(s) = \{s' \mid (s, s') \in \Phi_m\}$

If $\delta_m(s) = \varnothing$ and $s \notin S_m^f$, then s is a <u>dead</u> or <u>error</u> state.

Given two ANDFSM's A and B we wish to define what it means to say that A ***simulates*** B. Toward this end we define a function Ψ

Ψ: $S_A \to S_B$ where S_A is the state set of machine A and S_B is the state set of machine B

In general Ψ will be a many to one function. Armed with this function one is tempted to say that A simulates B if:

1. The states of A are in one to one correspondence with the states of B

2. Every legal transition in A has an image which is a legal transition of B:
   if $a \to a'$ then $\Psi(a) \to \Psi(a')$

3. Ψ maps the sets S_A^o onto S_B^o, and S_A^f onto S_B^f.

These conditions are far too restrictive for two major reasons.

First the simulating machine A may require several state transitions (steps) to simulate one step of B. So a one to one correspondence is impossible. Second, we are only interested in simulations of machines M that are functional, that is:

for any $s_1 \in S_m^o$, if $\exists$ sequences $(s_i^1)_{i=2}^{k1}$, $(s_i^2)_{i=2}^{k2} \ni$

i) $s_1 \to s_2^1 \to s_3^1 \to \ldots \to s_{k1}^1$
where
$s_{k1}^1 \in S_m^f$ the set of final states of M

and

$$s_1 \rightarrow s_2^2 \rightarrow s_3^2 \rightarrow \ldots \rightarrow s_{k2}^2$$

where

$$s_{k2}^2 \in S_m^f$$

then

$$s_{k1}^1 \equiv s_{k2}^2$$

We call a sequence of states as in i) a computation sequence. Consequently, we do not care if the machine doing the simulation of M can simulate all computation sequences of M but only if at least one can be simulated. This motivates the following definition.

DEFINITION

Let $M_A = < S_A, S_A^0, S_A^f, \Phi_A>$ and $M_B = <S_B, S_B^0, S_B^f, \Phi_B>$ be two ANDFSM's.

Let δ_A be a mapping δ_A: $S_A \rightarrow 2^S$ where

$\delta_A(s) = \{s' | (s,s') \in \Phi_A\}$ and δ_B the corresponding function for machine B.

Let Ψ be a many to one function

Ψ: $S_A \rightarrow S_B$

We say that M_A simulates M_B iff:

1. $\Psi(\delta_A(a)) \subsetneq \delta_B(\Psi(a)) \quad \forall a \in S_A$ (we have used the conventional abuse of notation, letting a set be an argument to Ψ)

2. $\exists$ k>0 $\exists$ $a_0, a_1, \ldots a_{k-1} \in S_A \ni$

$$(a_{i+1 (\mathrm{mod}\ k)} \in \delta_A(a_{i (\mathrm{mod}\ k)}) \ \forall\ i \geq 0) \Rightarrow$$

$$(\exists\ j) \ni [\ \Psi(a_{j+1 (\mathrm{mod}\ k)}) \in \delta_B(\Psi(a_j))]$$

3. $\delta_A(a) = \emptyset \Rightarrow \delta_B(\Psi(a)) = \emptyset$

4. $\forall\ b \in S_B^0 \ \exists\ a \in S_A^0 \ni \Psi(a) = b$

Informally we can interpret these conditions as:

1. If a transition is legal in machine A than the image of the transition in machine B is legal.

2. There are several interpretations for this clause. Most loosely the

condition states that "machine A makes progress". Alternately, M_A loops only if M_B loops, that is there is no infinite sequence of states of machine A that maps into a single state of machine B.

3. M_A hangs up only if M_B hangs up.

4. The start states of M_B are covered (under Ψ) by the start states of M_A.

This definition of simulation has one important shortcoming. The function Ψ is not restricted to be *semantics preserving*. We elaborate on this by way of an example. Suppose we had two nearly identical machines M_A and M_B. Suppose that the only difference was that we associate with the i^{th} final state of M_B the meaning "if halted in this state then the output value is i". On the other hand suppose all the final states of M_A have the meaning "if halted in this state the output value is π". Then it is easy to define a function Ψ such that one can prove M_A simulates M_B correctly. Nevertheless in every terminating simulation M_A will have an output which is different from M_B. Thus even though M_A simulates M_B it does not "compute the same thing". We can capture this idea formally by associating a semantic function $\mathcal{E}_M$ with a machine M where:

$$\mathcal{E}_M\colon S_M \rightarrow \text{"meanings"}$$

"Meanings" is an appropriate value domain that we wish to associate with the output states of M. Typically, "meanings" might be the set: integers $\cup$ strings $\cup$ arrays, etc. Let us call an ANDFSM M with an associated semantic function $\mathcal{E}_M$ a calculator and denote it by the pair $C = \langle \mathcal{E}_M, M \rangle$. We say that a calculator $C_A = \langle \mathcal{E}_A, M_A \rangle$ correctly implements a calculator C_B if

1. M_A simulates M_B

and

2. $\mathcal{E}_A(s) = \nu[\mathcal{E}_B(\Psi(s))] \quad \forall\ s \in S_A^f$.

where Ψ is an injective map from the domain of meanings of C_B to the domain of meanings of C_A.

3. $\forall \ s \in S_B^f \ \exists \ s' \in S_A^f \ \ni \ \Psi(s') = s$

i.e. the final states of C_B are covered by the final states of C_A.

With this precise definition of correct implementation we are equiped to prove correctness of certain machine architectures. We demonstrate this by the following example.

We introduce a class of programming schemas, the queued data flow schemas. This class of schemas is the same as the class of data flow schemata except that the links are queues and the firing rule is different:

1. A link node rather than holding at most one token, is an unbounded queue. Tokens are removed from the queue with a FIFO discipline.

2. An actor of a QDFS may fire iff all of its input links are non-empty. An actor fires by absorbing one token from each of its input links, and some finite but indeterminate time later, places one result token on its output link queue.

3. All actors of a QDFS are primitive computational functions.

4. The inputs to a QDFS are the set of links which have no predecessor actors - L_i.

5. The outputs of a QDFS are the set of links which have no successor actors - L_o.

6. A QDFS is said to be initialized iff

$\forall \ l \in L_i \ \ |l| = 1$

where $|l|$ denotes the length of the queue of link l

and

$\forall \ l \notin L_i \ \ |l| = 0$

7. A QDFS is said to be terminated iff

$\forall \ l \in L_o \ \ |l| > 0$ and no actor is enabled

and cleanly terminated iff

$\forall\ l \in L_o \quad |l| > 0$
$\forall\ l \notin L_o \quad |l| = 0$ and no actor is enabled

Let Π be a DDFS. We can model Π with a ANDFSM P. Before defining the states of P we introduce some notation.

A is the set of actors of Π
L is the set of links of Π
V is the domain of values of the tokens

we assume for convenience that the actors of Π are ordered and that we can refer to the i^{th} one as a_i. Similarly we number the links of Π and refer to the i^{th} one as l_i.

D_p is a function - D_p: A $\rightarrow$ OPERATORS

where o $\in$ OPERATORS is a six-tuple of the form $o = \langle opc, p_1, p_2, p_3, d_1, d_2 \rangle$ and

$D_p(a_i) = o$ iff p_i is the name of the i^{th} input link of a_i and d_i is the name of the i^{th} output link of a_i. opc $\in$ OPC is the name of the function that a_i computes. Thus we say that a_i computes the function f_{opc}: $V \times V^* \times V^* \rightarrow V$.

(V^* denotes V U {empty} where V is the domain of token values and empty $\notin$ V).

We assume for convenience that:

1. An actor has at most three inputs and that its output link has at most two output arcs. We will model an actor with a branching output link as an actor having two non-branching output links that receive the same values when the actor fires.

2. There exists a function NIN: OPC $\rightarrow$ {1,2,3} where NIN(opc) = i $\Rightarrow$ the function f_{opc} takes i non-empty input arguments.

3. p_j = nil for j > NIN(opc) (we use the reserved word nil to denote an absent link or argument)

4. There exists a function NOUT: A $\rightarrow$ {1,2} where

NOUT(a_i) = j $\Rightarrow$ a_i has j outputs

5. d_j = nil for j > NOUT(a_i)

We define the state of P as a pair $\langle D_p, S_p \rangle$ where:

D_p is as defined above

and

S_p: L ∪ nil → V^*

where

$S_p(l) = \langle v_1, v_2, \ldots v_n \rangle$ if the current contents of the queue of link l is (from tail to head) $v_1, v_2 \ldots v_n$.

= empty if queue l is empty or l = nil

We define a semantic function for P:

$\mathcal{E}_p(S_p)$ is a k-tuple where k = |L|

and

the i^{th} element of the k-tuple is $S_p(l_i)$.

It should be noted that for a given QDFS Π that the only state component of the corresponding machine P that is not constant is S_p. We will abuse the notation and refer to S_p as the state of P. (For this simple language, D_p is constant and might be better considered a semantic function. However in general D_p may not be constant, for example in languages with procedures, and thus has been included as part of the state.) Letting |l| denote the number of elements in the queue with name l, we define:

S_p is an initial state of P if

$\forall\ l \in L_i\ |S_p(l)| = 1$
$\forall\ l \notin L_i\ |S_p(l)| = 0$

(That is, all the input links have one token on them and all the other links are empty)

S_p is a final state if

$\forall\ l \in L_o\ \ |S_p(l)| > 0$
(every output link is nonempty)

and

$\nexists\ a_i \ni$
if
$D_p(a_i) = \langle opc, p_1, p_2, p_3, d_1, d_2 \rangle$
then
$S_p(p_i) \neq$ empty $\forall\ i \leq NIN(opc)$

i.e. there are no enabled actors

Lastly, we define the transition relation of P, Φ_p.

If $S = \langle D_p, S_p \rangle$ is a state of P with some $a_i \in A \ni$

$$D_p(a_i) = \langle opc, p_1, p_2, p_3, d_1, d_2 \rangle$$

and

$$\text{for } j \leq NIN(opc)\ S_p(p_j) \neq \underline{empty}$$

and $S' = \langle D_p, S_p' \rangle$ is a state of P $\ni$

$$S_p'(l) = S_p(l) \quad \forall\ l \notin \{p_1, p_2, p_3, d_1, d_2\}$$

$$S_p'(p_j) = -1{\downarrow}S_p(p_j)$$

and

$$1{\downarrow}S_p'(d_j) = S_p(d_j)$$

and

$$1{\uparrow}S_p'(d_j) = f_{opc}(-1{\uparrow}S_p(p_1), -1{\uparrow}S_p(p_2), -1{\uparrow}S_p(p_3))$$

Then $(S_p, S_p') \in \Phi_p$.

We have used some notation borrowed from APL for vector manipulations - ↓, ↑. The action of these operators is simply defined:

$i{\uparrow}\underline{empty} = \underline{empty}$ for $i \in \mathcal{N}$

$i{\downarrow}\underline{empty} = \underline{empty}$ for $i \in \mathcal{N}$

$1{\uparrow}\langle v_1, v_2, \ldots v_k \rangle = v_1 \quad k > 0$

$-1{\uparrow}\langle v_1, v_2, \ldots v_k \rangle = v_k \quad k > 0$

$1{\downarrow}\langle v_1, v_2, \ldots v_k \rangle = \langle v_2, v_3, \ldots v_k \rangle \quad k > 0$

$-1{\downarrow}\langle v_1, v_2, \ldots v_k \rangle = \langle v_1, v_2, \ldots v_{k-1} \rangle \quad k > 0$

$1{\downarrow}v = \underline{empty}$

$-1{\downarrow}v = \underline{empty}$

where we denote the k-tuple whose i^{th} component is v_i by $v_1, v_2, \ldots v_k$. Also, we use ";" to signify appending of tuples or scalars to tuples.

Now we have completely specified C_p. We now define another calculator which models the following machine:

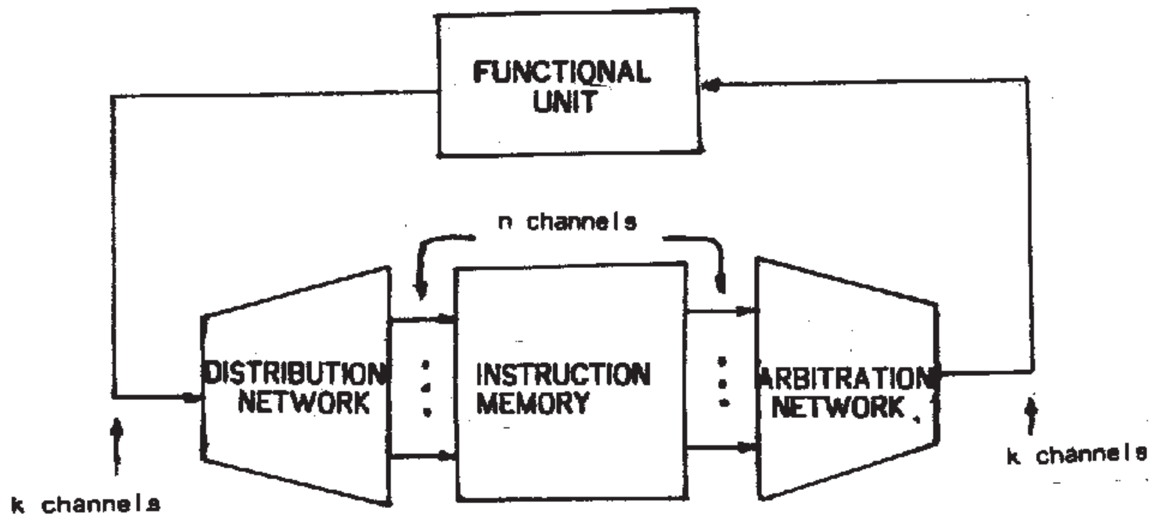

Figure 1
ELEMENTARY PROCESSOR

The elementary processor is a collection of four asynchronous modules - the instruction memory (IM), the arbitration network (AN), a set of functional units (FU), and the distribution network (DN). Each module is assumed to be connected (as shown in figure 1) to two of the other modules by a finite set of one way communications links called channels (arrows on the channels indicate the direction of data flow). A module may place a message into the input end of channel. This message is called a packet. Each channel is a buffer which connects two modules refered to generically as a source and a receiver, and has the following behaviour. The source examines the channel's status. If nothing is queued in the channel it is said to be empty and the source may place a packet in the channel. A receiver looks at the status of the channel. If a packet is queued in it, the receiver may remove it thus emptying the channel. Thus a channel is a buffer of length one which appears as an information sink to the source, and an information source to the receiver. Packets are received in the order in which they were transmitted and no packets are "lost". A channel buffer will be assumed to be part of the receiver module for the purposes of state definition.

The main module is the instruction memory. It consists of n identical units called cells. Each cell can be referred to by a name called a ***cell address.*** This name is simply an integer $0 \leq i < n$. Each of the cells is connected via a channel to an input of the arbitration network. The structure and function of the cell is simple. It consists of three registers and three unbounded queues. The contents of these six parts may be taken to be binary words of some finite length b for concreteness.

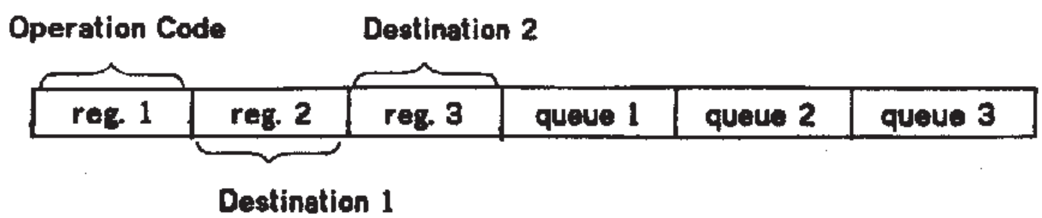

Figure 2

The first register contains a word referred to as an opcode - it is interpreted by the functional unit modules. The second and third registers contain queue names i.e. pairs of binary words whose first value is a number between 0 and n-1 specifying a cell, and whose second value is 1, 2, or 3 indicating a particular queue of the cell. To load an encoded program the contents of these registers are set initially by some outside agent. They remain static while the machine "runs". The three queues are FIFO buffers of unbounded length. They are referred to by a name of the form (i,j), where i is the name of the cell the queue is a part of, and $j \in \{1,2,3\}$ indicates which queue. The elements of the queue are members of some set of values B where B contains some suitable encoding of the set V.

The operation of the cells is simple. We assume that the EP has been constructed so that the elements of the set of opcodes OP is in one to one correspondence with the elements of OPC. For notational convenience we introduce the function

$$\text{OMAP}: \text{OP} \rightarrow \text{OPC}$$

We also assume that for any cell c that its queues q_j for $j > \text{NIN(OMAP(opcode))}$ are set to some distinguished state notused. The operation of the cell may be described as follows:

1. The cell's control examines its output channel. If it is not empty do step 1.

2. The cell's control examines its queues that are not in the state notused. If any of them are empty do step 2.

3. One value v_i is removed from the head of each queue. Then a string of binary words called an *operation packet* (*packet* for short) of the form $[c, \text{opcode}, v_1, v_2, v_3, d_1, d_2]$ is placed in the output channel where

   c is the cell name
   opcode is the contents of register one
   d_1 is the contents of register two
   d_2 is the contents of register three

   and

   v_i is the first element of the i^{th} queue for $i \leq \text{NIN(OMAP(opcode))}$

   and empty otherwise.

4. Do step 1.

The arbitration network receives packets from the IM and sends them to the FU. The AN's internal structure is not of interest. We assume its operation is:

1. Check if an output channel is empty. If not do step 1.
2. Check for a non-empty input queue. If none do step 2.
3. Select some non-empty input queue. Remove the packet, and place the packet in an empty output channel.
4. Do step 1.

We assume the AN is built so that when an input channel buffer b becomes non-empty at most j packets will be placed in the output channel before one is selected from channel b, for some $0 \leq j < \infty$. Furthermore, if any input channel is non-empty and any output channel is empty, then the AN must chose some packet to be placed in an empty output channel. Thus the arbiter implicit in the AN is "fair" in that an input packet is certain to be selected in some finite time. We ignore any additional details of the arbitration algorithm.

The FU is made up of a set of k identical units called ALU's. Each ALU is a small finite state machine which receives inputs from a channel connected to the AN, and sends outputs to the DN using a unique output channel. An ALU operates as follows:

1. Examine output buffer. If not empty do step 1.
2. Examine input queue. If empty do step 2.
3. Remove packet from input channel. The packet is of the form

$$\{i, opcode, v_1, v_2, v_3, d_1, d_2\}$$

4. Compute some value $v = f_{opcode}(v_1, v_2, v_3)$

5. Construct the *result packet* $\{i, v, d_1, d_2\}$

6. Place the packet in the output buffer.

7. Do step 1.

f_{opcode} is one of a set of transformations of the form f_0: $B \times B^+ \times B^+ \rightarrow B$
where
B is the set of binary words of length b
B^+ is the set B U {<u>empty</u>}

We will take the sets V and B to be identical for convenience. This saves us the bother of using a mapping function ψ every time we wish to relate a value in some queue of C_p (the schema's model) to C_m (the machines model). Further we assume that:

if opc = OMAP(opcode) then

$f_{opc}(x,y,z) = v$ iff $f_{opcode}(x,y,z) = v$

The distribution net is similar in function to the AN. We describe its operation briefly.

1. Examine the input queues. If all are empty do step 1.

2. Remove one packet $\{i,v,d_1,d_2\}$ from a non-empty input queue. Send packets conveying copies of the value v to queues d_1 and d_2. It is assumed that the destination cells will place the received value v on the tail of the specified queue.

3. Do step 1.

We now define the ANDFSM M which models the EP. A state of M is a quadruple S_m = <SIM,SAN,SFU,SDN> where

1. SIM corresponds to the state of the instruction memory. It is a function

SIM: I → NODES
where I is the set of cell names
NODES is a set of six-tuples
$SIM(i) = n = \langle opcode, q_1, q_2, q_3, d_1, d_2 \rangle$ where

opcode = contents of register one of cell i
q_j = contents of queue (i, j) $j = 1,2,3$
d_1 = contents of register 2 of cell i
d_2 = contents of register 3 of cell i

2. SAN corresponds to the state of the AN. It is an n-tuple where the j^{th} element of the tuple is:

   i) empty if the queue of the input channel j is empty

   ii) A tuple = $\langle i, opcode, v_1, v_2, v_3, d_1, d_2 \rangle$ if the queue of the j^{th} input channel contains the packet $p = [i, opcode, v_1, v_2, v_3, d_1, d_2]$.

3. SFU corresponds to the state of the functional unit. It is a k-tuple where the j^{th} element of the tuple is:

   i) empty if the queue of the input channel j is empty

   ii) A tuple = $\langle i, opcode, v_1, v_2, v_3, d_1, d_2 \rangle$ if the queue of the j^{th} input channel contains the packet $p = [i, opcode, v_1, v_2, v_3, d_1, d_2]$.

4. SDN corresponds to the state of the DN. It is a k-tuple where the j^{th} element of the tuple is:

   i) empty if the queue of the input channel j is empty

   ii) A tuple = $\langle i, v, d_1, d_2 \rangle$ if the queue of the j^{th} input channel contains the packet $p = [i, v, d_1, d_2]$.

The transition relation Φ_m is readily defined. We only consider the more interesting cases in detail.

1. State transition corresponding to a cell firing.

If $S_m = \langle SIM, SAN, SFU, SDN \rangle$
where
$\exists\ i \ni SIM(i) = \langle opcode, q_1, q_2, q_3, d_1, d_2 \rangle$
and
$|q_i| > 0$ for $i \leq NIN(OMAP(opcode))$
and
$SAN[i]$ = empty
(i.e. i is an enabled cell with an empty output channel)

and $S_m' = \langle SIM', SAN', SFU, SDN \rangle$
where

$$SIM(j) = SIM'(j) \qquad \forall\ j \neq i$$
$$SAN[j] = SAN'[j] \qquad \forall\ j \neq i$$

and

$SIM'(i) = \langle opcode, q_1', q_2', q_3', d_1, d_2 \rangle$ where
$q_j' = -1 \downarrow q_j$ for $1 \leq j \leq NIN(OMAP(opcode))$

$SAN'[i] = \langle i, opcode, -1\uparrow q_1, -1\uparrow q_2, -1\uparrow q_3, d_1, d_2 \rangle$

(i.e. cell i has transmitted the "appropriate" operation packet)
then
$(S_m, S_m') \in \Phi_m$

2. State transition corresponding to the AN firing.

If S_m = <SIM,SAN,SFU,SDN>
where
SAN[i] = <i, opcode, q_1, q_2, q_3, d_1, d_2> for some $0 \leq i < n$
SFU[j] = empty for some $0 \leq j < k$
(there is a packet queued on the AN and there is a free FU)
and
S_m' = <SIM,SAN',SFU',SDN>
where
SAN'[m] = SAN[m] $\forall$ m $\neq$ i
SFU'[m] = SFU[m] $\forall$ m $\neq$ j
SAN'[i] = empty
SFU'[j] = <i, opcode, q_1, q_2, q_3, d_1, d_2>
(the packet is forwarded to the free FU)
then
$(S_m, S_m') \in \Phi_m$

3. State transition corresponding to the FU firing.

If S_m = <SIM,SAN,SFU,SDN>
where
SFU[j] = <i, opcode, v_1, v_2, v_3, d_1, d_2> for some $0 \leq j < k$
SDN[j] = empty
(there is a "full" FU with a free output channel)
and
S_m' = <SIM,SAN,SFU',SDN'>
where
SFU'[j] = empty
SDN'[j] = <i, v, d_1, d_2>
and
$v = f_{opcode}(v_1, v_2, v_3)$
(the operation packet is removed and the result packet sent to the DN)
then
$(S_m, S_m') \in \Phi_m$

4. State transition corresponding to the DN firing.
This is analogous to case 2.

The set of initial states of M, S_m^0 is <SIM,SAN,SFU,SDN> where

SAN[i] = empty for $0 \leq i < n$
SFU[i] = empty for $0 \leq i < k$
SDN[i] = empty for $0 \leq i < k$

The set of final states S_m^f has the same restrictions on SAN, SFU, and SDN and in addition:

$\exists \;\; i \in I \;\; \ni \text{SIM}(i) = \langle \text{opcode}, q_1, q_2, q_3, d_1, d_2 \rangle$
and $|q_j| > 0 \;\; \forall \; j \leq \text{NIN(OMAP(opcode))}$
(i.e. no enabled cells)

Finally we define the semantic function for M, $\mathcal{E}_m$. It is a map from the state set of M onto the domain of meanings. We express it as a Curried function since it will be most useful in this form. In the definition of $\mathcal{E}_m$ we will capture the idea that the *semantics* of a state is determined by the contents of the queues. In order to preserve the correspondence (in the semantic functions) between queues and links we note that we must associate with a queue those values in the processor pipeline that are destined for the queue and those that originated at the queue.

$$\mathcal{E}_m\text{: } S_m \rightarrow [I \times \{1,2,3\} \rightarrow B \cup \{\underline{\text{empty}}\}] \quad \text{where}$$
$$\mathcal{E}_m(S_m)(i,j) = v = \langle X; \text{CON}(i,j); Y; Z \rangle$$

where

i) $X = v$ if $\exists \; m \ni \text{SDN}[m] = \langle s, v, d_1, d_2 \rangle$
and $d_l = (i,j)$ for $l = 1$ or 2
$= \underline{\text{empty}}$ otherwise

ii) $\text{CON}(i,j)$ = a tuple which is the contents of the $(i,j)^{th}$ queue
= <u>empty</u> if the $(i,j)^{th}$ queue is <u>notused</u>

iii) $Y = v$ if $\exists \; m \ni \text{SAN}[m] = \langle m, \text{opcode}, v_1, v_2, v_3, d_1, d_2 \rangle$
and $v = v_j$ and $d_l = (i,j)$ for $l = 1$ or 2
$= \underline{\text{empty}}$ otherwise

iv) $Z = v$ if $\exists \; m \ni \text{SFU}[m] = \langle s, \text{opcode}, v_1, v_2, v_3, d_1, d_2 \rangle$
and $v = v_j$ and $d_l = (i,j)$ for $l = 1$ or 2
$= \underline{\text{empty}}$ otherwise

This completes the definition of C_m.

We assume that we have built the models C_p and C_m so that the value domains V, and B are the same, b digit binary numbers. These sets as formal quantities are different, and a completely general treatment would require a map ν: V → B when we wish to compare elements in each set. We have intentionally made V and B "the same" so that we can avoid using ν when comparing elements of V and B and not be too abusive of the notation.

We now have completed the specification of our two models. In the proof of their equivalence we will find it useful to define a compiler from the machine P to M. A compiler will allow us to associate <u>any</u> state of P with an <u>initial</u> state of M. A compiler H from P onto M is a pair of one to one functions:

$HA: A \to I \times NODES$
$HL: L \to I \times \{1,2,3\}$
where
  $HL(l) = (i,j)$ where i is a cell name and j is a queue name
  $HA(a_i) = (i,n)$
where
  if $D_p(a_i) = \langle opc,\ p_1,\ p_2,\ p_3,\ d_1,\ d_2\rangle$ then
    $n = \langle OMAP^{-1}(opc), q_1, q_2, q_3, (id_1, rd_1), (id_2, rd_2)\rangle$
    where
      $q_j = S_p(p_j)$ if $p_j \neq \underline{nil}$
      $q_j = \underline{notused}$ otherwise
      $HL(d_1) = (id_1, rd_1)$
      $HL(d_2) = (id_2, rd_2)$

We assume that when a program is "loaded" into the EP the contents of the i^{th} cell is set to correspond to n where $\exists\ i \ni HA(a_i) = (i,n)$. Of course the machine must be at least as large as the programs loaded i.e. $|A| \leq |I|$. We are now ready to define $\Psi: S_m \to S_p$. We say that

$$S_p = \Psi(S_m) \quad \text{if } \forall\ l \in L$$

$$S_p(l) = w \text{ and } w = \langle X; CON[HL(l)]; Y; Z\rangle$$

where
  i) $X = v$ if $\exists\ m \ni SDN[m] = \langle n, v, d_1, d_2\rangle$ and $HL(l) = d_1$ or d_2

   $= \underline{empty}$ otherwise

  ii) $Y = v$ if $\exists\ i,j \ni SAN[i] = \langle i, opcode,\ v_1,\ v_2,\ v_3,\ d_1,\ d_2\rangle$ and $HL(l) = (i,j)$ and $v = v_j$

   $= \underline{empty}$ otherwise

  iii) $Z = v$ if $\exists\ j,m \ni SFU[m] = \langle i, opcode,\ v_1,\ v_2,\ v_3,\ d_1,\ d_2\rangle$ and $HL(l) = (i,j)$ and $v = v_j$

   $= \underline{empty}$ otherwise

We now prove that M simulates P by proving four lemmas. Each lemma will establish one of the four parts of the definition of simulation.

Lemma 1. Let S_m, S_m' be two successive states in a computation sequence of M starting from $S_m^0 = H(S_p^0)$. Then either

i) $\Psi(S_m) = \Psi(S_m')$
or
ii) $(\Psi(S_m), \Psi(S_m')) \in \Phi_p$

Proof (by induction on the number of transitions of machine M).

We assume we have a compiler as defined above, and that the machine is placed in the initial state as dictated by the function H. Then referring to the state of an ANDFSM existing <u>after</u> the j^{th} transition as the j^{th} state, and denoting this as S_m^j, we have:

$$\Psi(S_m^0) = S_p^0$$

and the lemma holds trivially after 0 transitions.

Assume that the lemma holds after l transitions (l>0). Thus

$$\Psi(S_m^l) = \Psi(S_m^{l-1}) \quad \text{or} \quad (\Psi(S_m^{l-1}), \Psi(S_m^l)) \in \Phi_p$$

We distinguish four cases for possible successor states of S_m^l:

i) Corresponding to a cell firing.

Then

$S_m^l = \langle SIM_l, SAN_l, SFU_l, SDN_l \rangle$

where

$\exists\ j \ni SIM_l(j) = \langle opcode, q_{1j}, q_{2j}, q_{3j}, d_{1j}, d_{2j} \rangle$

$SAN_l[j] =$ <u>empty</u>

and

$|q_{ij}| > 0$ for $1 \leq i \leq NIN(OMAP(opcode))$,

(i.e. cell j is enabled)

and

$S_m^{l+1} = \langle SIM_{l+1}, SAN_{l+1}, SFU_{l+1}, SDN_{l+1} \rangle$

where

$$SFU_l[k] = SFU_{l+1}[k] \quad \text{for } k \geq 0$$
$$SDN_l[k] = SDN_{l+1}[k] \quad \text{for } k \geq 0$$
$$SIM_l(k) = SIM_{l+1}(k) \quad \forall\ k \neq j$$
$$SAN_l[k] = SAN_{l+1}[k] \quad \forall\ k \neq j$$

and

$$SIM_{l+1}(j) = \langle j, opcode, q_{1j}', q_{2j}', q_{3j}', d_{1j}, d_{2j} \rangle$$

where

$$q_{ij}' = -1{\downarrow}q_{ij}, \quad i = 1,2,3$$

and

$$SAN_{l+1}[j] = \langle j, opcode,\ v_1,\ v_2,\ v_3,\ d_{1j},\ d_{2j} \rangle$$

where

$$v_i = -1\uparrow q_{ij}$$

(i.e. cell j fires sending an operation packet to the AN)
Let $S_p^z = \Psi(S_m^z)$ for any $z \in \mathcal{N}$. Clearly we have to examine only $S_p^l(k_i)$ and $S_p^{l+1}(k_i)$ for

$$k_i = HL^{-1}(j,i) \quad i = 1,2,3$$

since $\forall z \in L,\ z \neq k_i,\ S_p^l(z) = S_p^{l+1}(z)$
(i.e. the inverse image (under HL) of $l \notin \{k_i\}$ are unchanged)

let $S_p^l(k_i) = v_{1i};v_{2i};v_{3i}$ where v_{2i} is the contribution from the queue in the cell i.e. $v_{2i} = CON(HL(k_i))$, then

$$S_p^{l+1}(k_i) = v_{1i};v_{2i}';v_{3i}' \text{ where}$$

$$v_{2i}' = -1\downarrow v_{2i}$$
$$v_{3i}' = (-1\uparrow v_{2i});v_{3i}$$

by definition of Φ_m and Ψ

so that $v_{2i}';v_{3i}' = (-1\downarrow v_{2i});(-1\uparrow v_{2i});v_{3i} = v_{2i};v_{3i}$

therefore $\Psi(S_m^l) \equiv S_p^l = S_p^{l+1} \equiv \Psi(S_m^{l+1})$

so the lemma holds

ii) State change corresponding to the AN firing. This is analogous to case i) and is omitted.

iii)State change corresponding to the FU firing.
Then

$S_m^l = \langle SIM_l, SAN_l, SFU_l, SDN_l \rangle$ where $\exists$ a j $\ni$

$SFU_l[j] = \langle k, opcode, v_1, v_2, v_3, d_{1k}, d_{2k} \rangle$
and
$SDN_l[j] =$ <u>empty</u>
(i.e. there is a "full" FU with an empty output channel)
and

$S_m^{l+1} = \langle SIM_{l+1}, SAN_{l+1}, SFU_{l+1}, SDN_{l+1} \rangle$

where

$SIM_l(i) = SIM_{l+1}(i)$ for $i \geq 0$
$SAN_l[i] = SAN_{l+1}[i]$ for $i \geq 0$
$SFU_l[i] = SFU_{l+1}[i]$ $\forall i \neq j$
$SDN_l[i] = SDN_{l+1}[i]$ $\forall i \neq j$

and

$SFU_{l+1}[j] =$ <u>empty</u>
$SDN_{l+1}[j] = \langle k, v, d_{1k}, d_{2k} \rangle$

where

$v = f_{opcode}(v_1, v_2, v_3)$

(the FU has absorbed the input and sent its result packet to the DN)

As in case i) we need only consider $S_p^{l}(z)$, $S_p^{l+1}(z)$ for

$$z = HL^{-1}(s_i) \quad s_i = (k,i) \quad i = 1,2,3$$

and

$$z = HL^{-1}(d_{1k}),\ HL^{-1}(d_{2k})$$

since $\forall\, z \in L,\ z \notin \{s_1, s_2, s_3, d_{1k}, d_{2k}\}\ \ S_p^{l}(z) = S_p^{l+1}(z)$

Let $v_i = S_p^{l}(s_i)$

(we note that by the definition of Ψ, $v_i \neq$ empty for $i \leq$ NIN(OMAP(opcode)) since SFU[j] $\neq$ empty. This condition on v_i together with the definition of H allows us to conclude that the actor a which was mapped to cell k has only non-empty input links)

then

$$S_p^{l+1}(s_i) = v_i' = -1\downarrow v_i \quad \text{by definition of } \Psi \text{ and } \Phi_m$$

similarly, let $w_i = S_p^{l}(d_{ik})$ then

$$S_p^{l+1}(d_{ik}) = w_i' = v;w_i$$

but

$$\begin{aligned} v &= f_{opcode}(v_1, v_2, v_3) = f_{opc}(v_1, v_2, v_3) \\ &= f_{opc}(-1\uparrow S_p^{l}(HL^{-1}(k,1)), -1\uparrow S_p^{l}(HL^{-1}(k,2)), -1\uparrow S_p^{l}(HL^{-1}(k,3))) \end{aligned}$$

where $opc = 1\uparrow D_p(a)$

But then by definition of Φ_p, $(S_p^{l}, S_p^{l+1}) \in \Phi_p$

Whence $(\Psi(S_m^{l}), \Psi(S_m^{l+1})) \in \Phi_p$ since

$$\Psi(S_m^{l}) \equiv S_p^{l} \quad \text{and} \quad \Psi(S_m^{l+1}) \equiv S_p^{l+1}$$

so the lemma holds.

Notice that the above argument does not hold if d_{1k} or $d_{2k} = (k,n)$ for some n (i.e. an actor is its own successor) since then $v_n' \neq -1\downarrow v_n$. The proof of this special case is left to the reader.

iv) A state change corresponding to the DN firing. This is just like case i) and so is omitted.

∎

The proof of the second condition is a bit more brief.

LEMMA 2. $\exists\, k>0 \ni \exists\, s_0, s_1, \ldots s_{k-1} \ni$

$$(s_{i+1 (\text{mod } k)} \in \delta_m(s_{i (\text{mod } k)}) \quad \forall\, i \geq 0 \Rightarrow$$
$$(\exists\, j) \ni [\Psi(s_{j+1 (\text{mod } k)}) \in \delta_p(\Psi(s_j))]$$

Proof:

We note that the existence of such a cycle of states implies that M may "loop" forever. Let us restrict our attention to the states of M that comprise this loop. Let k be the number of states in the loop and call the i^{th} state in the loop s_i. Then for the states of the loop

$$\forall\ i \geq 0 \quad s_{i+1 (\mathrm{mod}\ k)} \in \delta_m(s_{i(\mathrm{mod}\ k)})$$

and the hypothesis is satisfied for this k and states $Z = \{s_i\}_0^{k-1}$. It is obvious that any such loop of states of M must include a change SFU (i.e. a functional unit must fire). Let $S_m^z \in Z \subsetneq S_m$ be the state prior to a change in SFU. Then by lemma 1

$$(\Psi(S_m^z), \Psi(S_m^{z+1})) \in \Phi_p$$

therefore

$$\Psi(S_m^{z+1}) \in \delta_p(\Psi(S_m^z))$$

We note that we can choose a labelling of the states of Z so that $z < k$. Whence the lemma follows immediately with $j = z$.

∎

The final two lemmas depend heavily on the properties of the compiler H.

LEMMA 3. $\delta_m(s) = \emptyset \Rightarrow \delta_p(\Psi(s)) = \emptyset$

Proof:
The key to the proof is that H is one to one.

$\delta_m(s) = \emptyset$ ⇒ there are no packets in the channels
⇒ the image of the state of M is totally determined by SIM (i.e. the contents of the IM)

therefore $\delta_m(s) = \emptyset \Rightarrow$ no enabled cells

whence by the definition of H there are no enabled actors.
We conclude that $\delta_p(\Psi(s)) = \emptyset$.

∎

Finally we have

LEMMA 4. $\forall\ s_p \in S_p^0\ \exists\ s_m \in S_m^0 \ni \Psi(s_m) = s_p$.

Proof: Follows trivially from the definition of H.

∎

From these four lemmas we may conclude that:

Theorem 1. The machine M correctly simulates any properly sized QDFS (with the restrictions on the actors as outlined above).

Now we wish to show that the machine M computes the same thing as P i.e. C_m correctly implements C_p. Theorem 1 proves that the part of the definition requiring correct implementation is satisfied. Notice that there is one small technical difficulty in the proof of the second part. It is that the quantities $\mathcal{E}_m(S_m)$ and $\mathcal{E}_m(S_p)$ are tuples of different lengths in general. We will say they are equal if and only if:

$$\text{For } s_p \in S_p \text{ and } s_m \in S_m \quad \mathcal{E}_m(s_m) = \mathcal{E}_p(s_p) \text{ iff}$$

$$\forall \; i \ni l_i \in L, \quad \mathcal{E}_m(s_m)(HL(l_i)) = \mathcal{E}_p(s_p)[i]$$

We prove the second part:

LEMMA 5. $\forall \; s \in S_m \quad \mathcal{E}_m(s) = \mathcal{E}_p(\Psi(s))$

Proof: (by induction on the number of transitions)

Notice that the lemma proves a stronger property than required by the definition. This is useful in the induction proof since the induction hypothesis is now stronger.
After 0 transitions, calling the state of M, s_0 we have:

$$\mathcal{E}_m(s_0) = \mathcal{E}_p(\Psi(s_0))$$

since

$$\mathcal{E}_p(S_p)[i] \equiv S_p(l_i)$$

and

$$\mathcal{E}_m(s_0)(HL(l_i)) \equiv S_p(l_i)$$

by definition of $\mathcal{E}_m$, $\mathcal{E}_p$, Ψ and H.

Assume that the hypothesis holds after h > 0 transitions. Then just as in lemma 1 there are four cases to be considered. Again only the case of an FU firing is non-trivial and so is the only one presented here.

case i) State transition corresponding to a cell firing.

case ii) State transition corresponding to the AN firing.

case iii) State transition corresponding to the DN firing.

case iv) State transition corresponding to the FU firing (i.e. SFU, SDN changes).

Let S_m^h = <SIM_h, SAN_h, SFU_h, SDN_h> be the state after the h^{th} transition then

$$\exists \text{ an } n \ni SFU_h[n] = \langle i, opcode, v_1, v_2, v_3, d_1, d_2 \rangle$$

and

$$SDN_h[n] = \underline{empty}$$

and

$$\mathcal{E}_m(S_m^h) = \mathcal{E}_p(\Psi(S_m^h))$$

after the $(h+1)^{st}$ transition $S_m^h \equiv S_m^{h+1}$ <u>except:</u>

$$SFU_{h+1}[n] = \underline{empty}$$
$$SDN_{h+1}[n] = \langle i, v, d_1, d_2 \rangle$$

where

$$v = f_{opcode}(v_1, v_2, v_3)$$

Clearly then $\mathcal{E}_m(S_m^{h+1})(HL(l_i)) = \mathcal{E}_p(\Psi(S_m^{h+1}))[l]$
$\forall$ i, $HL(l_i) \notin \{(i,1), (i,2), (i,3), d_1, d_2\}$

For $p_j = (i,j)$ $j = 1,2,3$

$$\mathcal{E}_m(S_m^{h+1})(p_j) = -1\downarrow w_j \text{ where } w_j = \mathcal{E}_m(S_m^h)(p_j) \text{ by definition of } \Phi_m$$

$$\Psi(S_m^{h+1})(HL^{-1}(p_j)) = -1\downarrow w_j \text{ by definition of } \Psi, \text{ HL and lemma 1}$$

therefore letting $l_q = HL^{-1}(p_j)$

$$\mathcal{E}_p(\Psi(S_m^{h+1}))[q] = -1\downarrow w_j = \mathcal{E}_m(S_m^{h+1})(p_j)$$

by definition of $\mathcal{E}_p$, and Φ_p

For d_j , $j = 1,2$

$$\mathcal{E}_m(S_m^{h+1})(d_j) = v;\mathcal{E}_m(S_m^h)(d_j) \text{ by definition of } \Phi_m \text{ and } \mathcal{E}_m.$$

now

$$\Psi(S_m^{h+1})(HL^{-1}(d_j)) = v;\Psi(S_m^h)(HL^{-1}(d_j)) \text{ by definition of } \Psi \text{ and HL}$$

so that letting $l_q = HL^{-1}(d_j)$

$$\mathcal{E}_m(S_m^{h+1})(d_j) = \mathcal{E}_p(\Psi(S_m^{h+1}))[q] \text{ by definition of } \mathcal{E}_p,$$

and induction assumption.

so the lemma holds.

Notice that the proof above does not hold if an actor is its own predecessor. The proof of this special case is left to the reader.

∎

We observe that the third requirement for the correctness of an implementation (covering of final states) is trivially true. Thus we state without proof

LEMMA 6. $\forall s_p \in S_p^f \; \exists s_m \in S_m^f \ni \Psi(s_m) = s_p$

Proof: Obvious from the definition of $\bar{M}$ and final states.

∎

Thus we have

Theorem 2. The machine M correctly implements any properly sized QDFS P.

Proof: Immediate from theorem 1 and lemmas 5 and 6.

∎

We notice that for any QDFS P in its *initial* state, no directed cycle in P has any tokens on its component links. It is well known that for such schemas, the token load of a link during "execution" is at most one [5], [6]. Thus we may conclude

LEMMA 7. If we only compile QDFS's that are in an initial state onto an EP, then during the execution of the schema on the EP, the cells queues have length of at most one.

Proof:

Let P be the machine which models a QDFS that is in an initial state S_p^0. Let M be placed in a start state corresponding S_p^0. By theorem 2, M will correctly simulate a computation of P. More important, by lemma 5 after any step of the simulation (with M in state S_m^j)

$$\mathcal{E}_m(S_m^j) = \mathcal{E}_p(\Psi(S_m^j))$$

But the i^{th} component of $\mathcal{E}_m(S_m^j)$ is a tuple whose length bounds the length of some queue q in a cell of the instruction memory of the elementary processor, by construction of C_m and $\mathcal{E}_m$. We may conclude that for any q

$$\exists \text{ a pair } p \ni \; |q| \leq |\mathcal{E}_m(S_m)(p)|$$

(note the length of q is zero if q is <u>notused</u> or q = <u>empty</u>)

But either

$$\mathcal{E}_m(S_m^j)(p) = \mathcal{E}_p(\Psi(S_m^j))[i] \quad \text{for some } i$$

so that

$$|\mathcal{E}_m(S_m^j)(p)| = |\mathcal{E}_p(\Psi(S_m^j))[i]|$$

or p has no image (under HL^{-1}) in P and hence is <u>notused</u>

so that

$$|\mathcal{E}_m(S_m^j)(p)| = 0$$

but $\forall\ l_i \in L,\ \mathcal{E}_p(\Psi(S_m^j))[i]$ = contents of the i^{th} link of the schemas P models, by construction.

therefore by the above observation we may conclude that

$$|\mathcal{E}_p(\Psi(S_m))[i]| \leq 1$$

whence

$$|q| \leq 1 \quad \text{for any queue of the IM}$$

∎

We conclude that the machine EP correctly executes QDFS's (starting from an initial state) even if the queues of the cells are replaced by single word registers. Further we notice that no component of the EP ever used the first element of the packets that were placed into the channels which was simply the originating cells name. (This component was only introduced to simplify the notation). Consequently, we may eliminate this part of all packets without affecting the above results. Doing these things we see that the resulting machine is simply the <u>e</u>lementary <u>d</u>ata <u>f</u>low <u>p</u>rocessor. Since the class of QDFS's is at least as large as the class of WFDFS's composed only of primitive computational functions (call them <u>S</u>impleWFDFS's) we conclude that:

Theorem 3. The EDFP correctly implements the class of properly sized and initialized SWFDFS's.

∎

We note several things about the proof. First, though the level of detail was rather high, the proof was straightforward and required no "tricks". Second the modular structure of the machine was exploited in the proof in several ways. Most significantly, it allowed fracturing the definition of S_m and Φ_m and the main lemmas (one and five) into several disjoint subcases. I suspect that the extension of this proof technique to more complicated packet communication architectures will therefore be straightforward.

This paper was inspired by the work of J. Rumbaugh [9] in which he proved his machine for executing data flow programs was correct. However the approach toward proof of correctness of an architecture in this paper differs from his in several important ways. First, Rumbaugh rather than using ANDFSM's as the basic model in his proofs, used a structure he called a non-deterministic information structure model (NDISM). The model has no more power than the ANDFSM's and the introduction of it is unnecessary. It leads one into a style of proof that appears rather ad-hoc and rather informal as opposed to the more firmly founded style of automata theory. Second, Rumbaugh fails to restrict his function Ψ to be semantics preserving in his definition of *simulation*. He does not explicitly address this problem, although the specific way in which he defines his function Ψ, it is semantics preserving. Third, the NDISM Rumbaugh uses to model his machine (BM) does not reflect the actual machine's (RM) operation accurately. There are many components of the RM that contribute to its state that do not figure in the state definition of the BM. He covers these discrepancies by a number of lemmas wich prove additional properties of the RM. I believe that his proof is fundementally correct, though it is always dangerous to follow a proof strategy which will allow you to derive consistent theorems that prove something other than what is desired. Omission of any one of these ad-hoc lemmas would have left a collection of consistent statements about the properties of the RM but would have fallen short of a correctness proof of the BM in an unobvious manner. There were at least two reasons why this strategic error was made. First, was the absence of an adequate definition of what it means for a machine to be correct (correct simulation clearly being insufficient). Second was the failure to keep distinct the properies and behaviour of the actual machine (the RM) and the model of the machine (the BM).

The lemmas should have been formal statements about the behaviour of the BM and the correspondence between the BM and the RM been so close that the reader would believe that the properties held for the real machine. Although it is an unpleasant thought, one cannot construct a formal, rigourous proof about a *real* machine. Any formal proof about some property of a real machine must necessarily deal with a model of the machine. Thus it is imperative that the class of models chosen be simple so that credible modelling can be done. Though the class of models Rumbaugh chose were simple, he neglected to construct a model which accurately reflected the real machine. His proof is consequently less formal and more ad-hoc then it might have been.

It is hoped that the proof presented in this paper avoids at least some of the aforementioned pitfalls. However it has several pitfalls of its own. The most significant is that the level of abstraction achieved by the models is very low. Consequently, they reflect much of the detailed structure of that which is being modelled. Although the model of the QDFS's is probably acceptable, the model of the EP is rather involved. Hopefully, one can achieve higher levels of abstraction, otherwise proofs for complex machines will become unwieldy. Dave Ellis' work may shed some light on this problem.

Currently, work is proceding on a proof of correctness for a more elaborate data flow machine. The machine has the same general structure as outlined in CSG memo 138 [7]. Since the machine has much greater capabilities (procedures, conditionals etc.) the proof is longer. It is no more complicated, but the characterization of the states of the machine is much more involved and adds many "cases" to the proof. Aside from this, work seems to be proceeding much along the lines in this paper. Notable improvements include better handling of unused links and outputs in the models. Also greater care will be taken in keeping the properties of the models distinct from that which is being modelled.

ACKNOWLEDGEMENTS

I wish to thank Jack Dennis for challenging me to look at this problem, and for his continued interest, and Dave Ellis who provided some useful criticism. I also wish to thank Albert Meyer, who contributed several helpful ideas.

BIBLIOGRAPHY

1. Dennis, J., Fosseen, J., "Introduction to Data Flow Schemas". CSG Memo 81, Department of Elec. Eng. and Comp. Sci, MIT, Cambridge, MA.

2. Dennis, J., Misunas, D., "The Design of a Highly Parallel Computer for Signal Processing Applications", MAC TR101, Department of Elec. Eng. and Comp. Sci, MIT, Cambridge, MA. August 1974

3. Dennis, J., Misunas, D., "A Preliminary Architecture for a Basic Data Flow Processor" , MAC TR102, Department of Elec. Eng. and Comp. Sci, MIT, Cambridge, MA. August 1974

4. Fosseen, J., "Represention of Algorithms by Maximally Parallel Schemata", S.M. Thesis, Department of Elec. Eng. and Comp. Sci, MIT, Cambridge, MA. June 1972

5. Hack, M., "Analysis of Production Schemata", MAC TR94, Department of Elec. Eng. and Comp. Sci, MIT, Cambridge, MA. February 1972

6. Leung, C., "Formal Properties of Well-formed Data Flow Schemas", MAC Technical Memorandum 66, Department of Elec. Eng. and Comp. Sci, MIT, Cambridge, MA. June 1972

7. Miranker, G., "Implementation Schemes for Data Flow Procedures", CSG memo 138, Department of Elec. Eng. and Comp. Sci, MIT, Cambridge, MA. May 1976

8. Weng, K., "Stream Oriented Computation in Recursive Data Flow Schemas", MAC Technical Memorandum 68, Department of Elec. Eng. and Comp. Sci, MIT, Cambridge, MA. October 1975

9. Rumbaugh, J., "A Parallel Asynchronous Computer Architecture For Data Flow Programs", MAC Technical Memorandum 150, Department of Elec. Eng. and Comp. Sci, MIT, Cambridge, MA. May 1975