PB-157

B.H. Mayoh: Program Models: Meaning and Proof

TRYK: DAIMI/RECAU

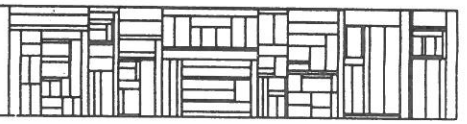
ISSN 0105-8517

# PROGRAM MODELS: MEANING AND PROOF

Brian H. Mayoh

DAIMI PB-157
January 1983

Computer Science Department
**AARHUS UNIVERSITY**
Ny Munkegade – DK 8000 Aarhus C – DENMARK
*Telephone: 06 – 12 83 55*



# PROGRAM MODELS: MEANING AND PROOF

Brian H. Mayoh

Abstract

When does one understand a program P? One reasonable answer is: when one has a conceptually clear model M such that any relevant question about P can be reformulated as a logically equivalent question about M. This paper argues for diagram models for parallel programs as a simple generalization of the familiar flowchart, graph and equation-set models of sequential programs. Diagram models of Petri Nets were introduced by Hinderer in [Hi].

# PROGRAM MODELS: MEANING AND PROOF

When does one understand a program P? One reasonable answer is: when one has a conceptually clear model M such that any relevant question about P can be reformulated as a logically equivalent question about M. This paper argues for diagram models for parallel programs as a simple generalization of the familiar flowchart, graph and equation-set models of sequential programs. Diagram models of Petri Nets were introduced by Hinderer in [Hi].

In section 1 flowchart and graph models of sequential programs without recursion are discussed; in section 2 equation-set models are introduced as a way of handling recursion; in section 3 diagram models are introduced as a way of handling parallelism; in section 4 we show how the "true concurrency" of Petri nets and the "synchronizations" of Milner agents are captured by diagram models; in section 5 the connection with Scott's new theory of domains is mentioned. The first two sections contain much familiar material, but they lay the necessary groundwork for diagram models and they illustrate the essential requirements on any program model

- to allow proofs of relevant assertions about programs;
- to reflect the static structure of the program text;
- to capture the dynamic behaviour of the program.

We should also justify our rejection of the slogan: a program is its own best model. In the axiomatic approach to the semantics of programming language, one has a special logic with symbols for programs. Axiomatic semantics started with Hoare [Ho1] and is still popular in the form of dynamic logic [Pr], algorithmic logic [Sa], temporal logic [Pn] and predicate transformer semantics [D]. Since one can understand a program without learning a new logic, there is a need for a semantics which assigns a model to a program. The model of a program should be such that one can use ordinary everyday logical arguments to prove relevant true assertions about the program. The four kinds of models in this paper sound very operational, but the distinction between denotational semantics [St] and operational semantics [Pl] is very thin, and there is an equation-set model behind every denotational semantics.

## #1 Sequential programs without recursion

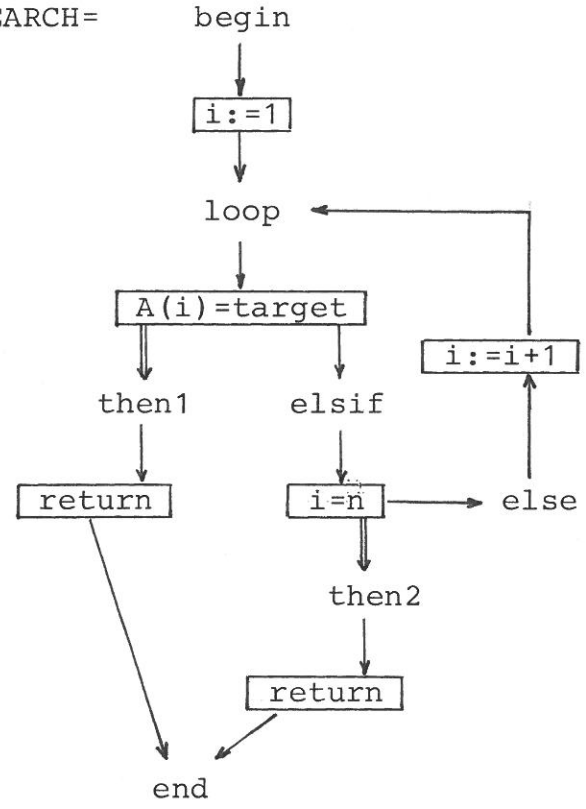
The usual way of understanding a sequential program is to convert it to a flow chart. For us a flow chart consists of

- labelled frames
- labelled arrows from frames to nodes
- arrows from nodes to frames

The nodes in a flowchart correspond to control points in a program, the labelled frames correspond to program statements, and the arrows give the flow of control in the program. Usually there is only one arrow from each node, but there is no reason to insist on this and forbid non-deterministic programs.

Example

SEARCH=




```
procedure Search;
  begin                          -- node
        i:=1;
   loop                          -- node
     if A(i) = target
     then                        -- node
        return(i);
     elsif i=n
     then                        -- node
        return(0)
     else                        -- node
        i:=i+1
     end if
   end loop
end Search;
```

□

Once we have the flowchart model of a program P any question about P can be answered by induction [F]

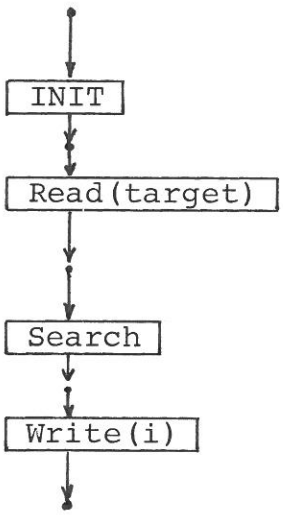
- assign assertions to the nodes in the flowchart
- prove that frames preserve the truth of the assertions.

This proof method is sound because computations of a sequential program without recursion correspond to paths through its flowchart model.

Flowcharts are adequate models for many programs because there are several natural combinators for building new flowcharts from old. Let us describe the substitution combinator that is useful for modelling nested blocks and nonrecursive procedures. When presenting a flowchart, one can use frames whose label names another flowchart. If the labelled frame has the same number of source and sink nodes as the named flowchart, then the labelled frame can be replaced by the named flowchart.

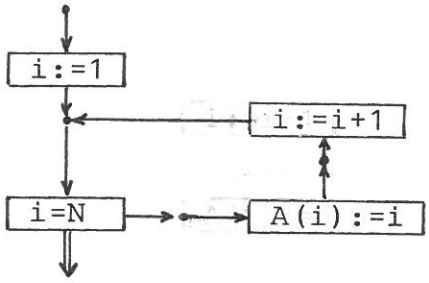
Example

MAIN=




```
begin                    -- MAIN Program
     begin               -- INIT Block
         for i in 1..N
         loop A(i).=i
         end loop;
     end;          -- INIT Block
     Read(target);
     Search;
     Write(i);
end                      -- MAIN Program
```

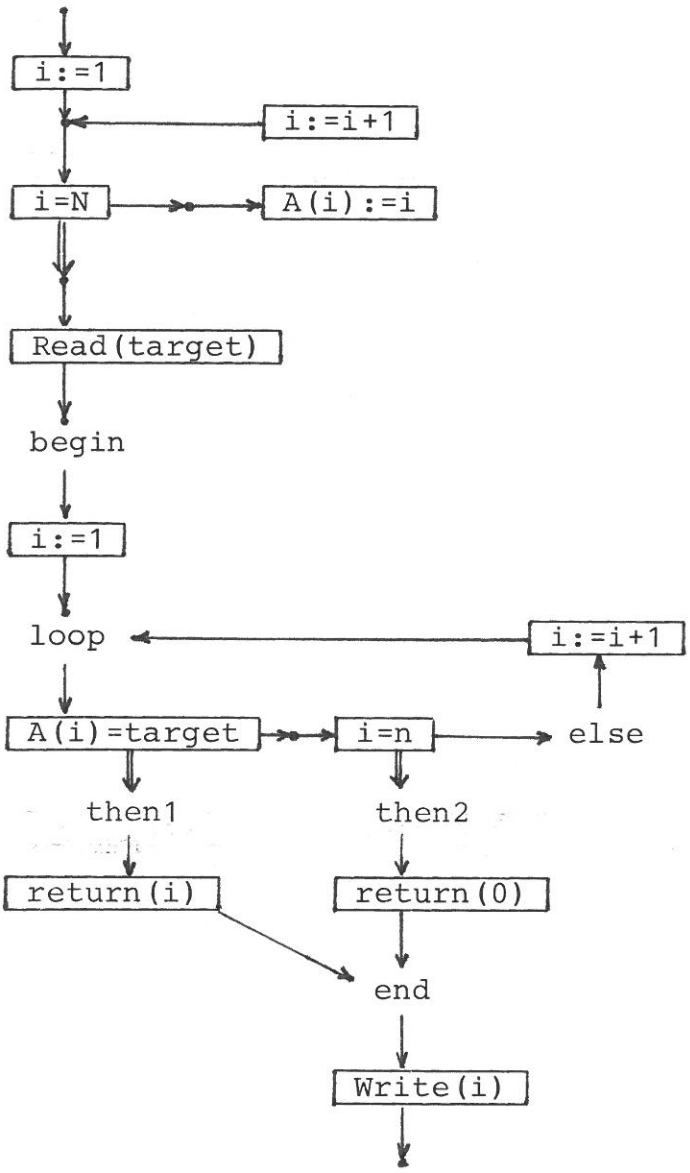
INIT=




SEARCH flowchart given earlier.

□

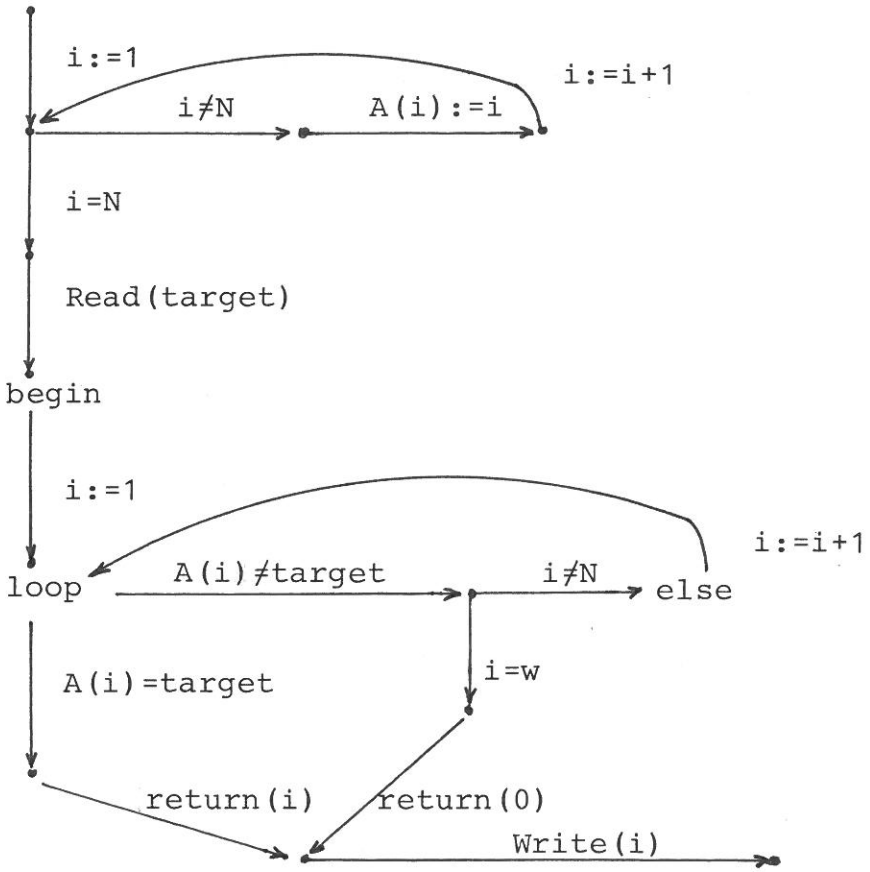
The result of the substitution is




One of the disadvantages of flowcharts is they become very bulky, but converting flowcharts to graphs by labelling all arrows and dropping frames gives more concise models. For us a graph consists of labelled vertices and labelled directed edges; in a graph model of a program the vertices correspond to control points and the edges correspond to program statements.

Example

The program whose flowchart model was given in the last example has the graph model:




Notice how tests and other frames with several exits give several edges in graph models. Sometimes labelling such edges is problematic. □

Once we have the graph model of a program P, any question about P can be answered by induction

-- assign assertions to the vertices of the graph

-- prove that edges preserve the truth of the assertions.

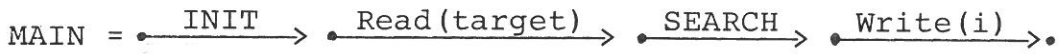
This proof method is sound because computations of a sequential program without recursion correspond to paths through its graph model.

Graphs give adequate models for programs in many programming languages because the extensive theory of graph grammars [Gr, Pa] gives many natural combinators for building new graphs from old. Let us describe the edge replacement combinator that is useful

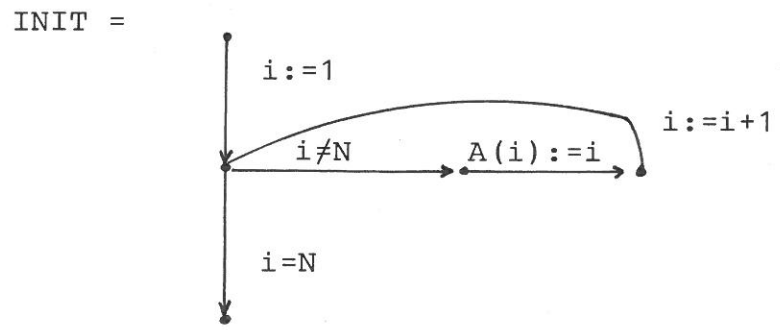
for modelling nested blocks and non-recursive procedures. When presenting a graph one can use edges whose label names another graph. If the named graph has one source vertex and one sink vertex, then the labelled edge can be replaced by the named graph.

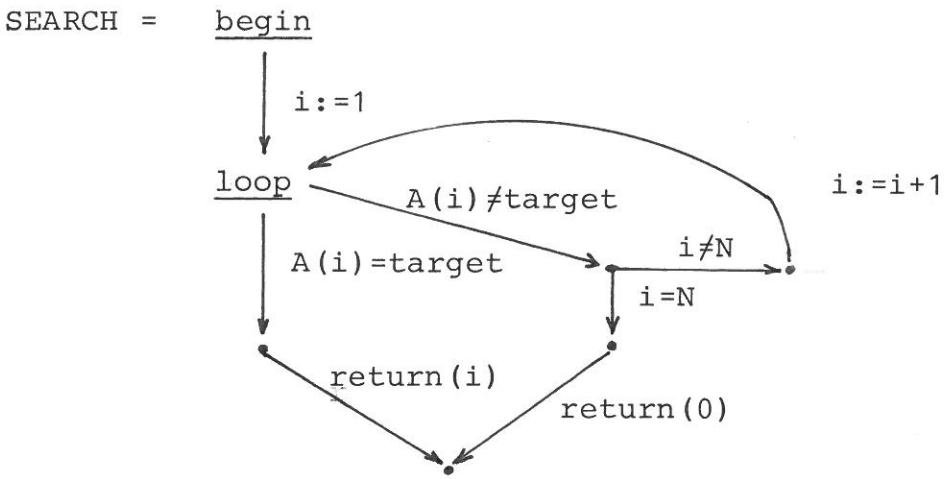
Example

The graph in the last example is the result of taking the graph




and replacing edges labelled INIT and SEARCH by




□

One should not underestimate the expressive power of graph models. In the form of transition systems they have been used for many years to capture the dynamic behaviour of parallel processes [Ke] and they have recently become popular in operational semantics [Pl]. The only reason for considering other kinds of program models is the need for models that directly reflect static program texts, not the dynamic behaviour of programs.

## #2 Sequential programs with recursion

Consider the flow chart or graph model of a recursive procedure P. The flowchart model contains a frame labelled P and using the substitution combinator of the last section leads to an infinite regress; the graph model contains an edge labelled P and using the edge relabelling combinator also leads to an infinite regress; there is no finite flowchart or graph that captures the potential depth of recursion of P. In other areas of computer science - program schemes [Gu], abstract data types [ADJ], term rewriting systems [Hu], attribute grammars [CFZ] - sets of equations have helped in the handling of recursion, so they might provide suitable models for sequential programs with recursion. For us an equation-set is a set of pairs $(L_0,R_0)(L_1,R_1)\ldots$ where $L_0,L_1,\ldots$ are variables, $R_0,R_1,\ldots$ are sums of terms, and terms are action-variable pairs. In an equation-set model of a program, variables correspond to control proints and actions correspond to program statements.

Example

Earlier we have given a flowchart model and a graph model for:

```
procedure Search;
begin
     i:=1;
     loop
       if A(i) = target
       then return(i)
       elsif I=N
       then return(0)
       else i:=i+1
       endif
     end loop;
end Search
```

Each of these corresponds to two equation set models:

BACKWARD SEARCH

begin = [i:=1]: loop
loop = [A(i) = target]: $then_1$
+ [A(i) ≠ target]: elsif
$then_1$ = [return(i)]: end
elsif = [i=N]: $then_2$ + [i≠N]: else
$then_2$ = [return(0)]: end
else = [i:=i+1]: loop

FORWARD SEARCH

loop = [i:=1]: begin + [i:=i+1]: else
$then_1$ = [A(i) = target]: loop
elsif = [A(i) ≠ target]: loop
$then_2$ = [i=N]: elsif
else = [i≠N]: elsif
end = [return(i)]: $then_1$ + [return(0)]: $then_2$

□

Once we have the equation set model of a program P, any question about P can be answered by induction

- assign assertions to the variables in the equation set
- prove the validity of the equations for these assertions.

If the programming language has a predicate transformer semantics, it determines whether the forward or backward equation set should be used in formal proofs; if the programming language assigns state transition functions to statements (direct denotational semantics), the forward equation set should be used in formal proofs; if it assigns continuation transition functions to statements (continuation denotational semantics), the backward equation set should be used.

The astute reader has realised that equation sets are just another representation for graphs. However equation sets have the advantage that there is a canonical way of solving equation sets once assertions have been given to the parameters, the variables in the right sides of equations but not on the left side of any equation. The canonical solution is given by:

(Step 1) assigning true to variables that are not parameters;
(Step 2) calculating the assertion $a_i$ for each right side $R_i$;
(Step 3) assigning the assertion $a_i$ to the left side variable $L_i$;
(Step 4) if progress then goto Step 2;

in other words, by taking the least fixed point. Why is such a canonical solution useful? Consider an equation like

$$a = [E]: b$$

where $E$ is the name of some equation-set. When calculating the assertion for a from that for b according to Step 2 we can use the canonical solution for $E$. In the case when the equation $a = [E]:b$ is in the equation-set $E$ this still works and we avoid the infinite regress of the flowchart and graph models of recursive programs.

The last paragraph should not be taken as an argument against a substitution combinator for building new equation-sets from old. Such a combinator is essential for the syntax directed construction of the equation-set model of a program while it is being parsed.

If $E$ is the name of a set of equations with n parameters and x is the left side of one of these equations, then we allow the term $[E]_x$: $b_1\ b_2\ \ldots\ b_n$ to appear in the presentation of another equation set $E'$. One gets the equation-set $E'$ from the presentation using $[E]_x$: $b_1 \times b_2 \times \ldots \times b_n$ by

- replacing all the leftside variables in $E$ by new variables;
- replacing all the parameters in $E$ by $b_1, b_2, \ldots, b_n$ (we suppose an order on the parameters);
- replacing the term $[E]_x$: $b_1 \times b_2 \times \ldots \times b_n$ by the new variable for x;
- $E'$ is the resulting set of equations.

Example

Earlier we gave the flowchart and graph models for a program MAIN and an internal block INIT. The corresponding backward equation-set models are

```
M1 = [INIT]I1: M2                    I1 = [i:=1]: I2
M2 = [Read(target)]: M3              I2 = [i=N]: I5 + [i≠N]: I3
M3 = [SEARCH]begin: M4               I3 = [A(i):=i]: I4
M4 = [Write(i)]: M5                  I4 = [i:=i+1]: I2
```

Using the substitution combinator and the backward equation-set for SEARCH gives

```
M1      = I1'                        I1' = [i:=1]: I2'
M2      = [Read(target)]: M3         I2' = [i=N]: M2 + [i≠N]: I3
M3      = begin'                     I3' = [A(i):=i]: I4'
M4      = [Write(i)]: M5             I4' = [i:=i+1]: I2'
begin'  = [i:=1]: loop'
loop'   = [A(i)=target]: then1' + [A(i)≠target]: elsif
then1'  = [return(i)]: M4
elsif'  = [i=N]: then2' + [i≠N]: else'
then2'  = [return(0)]: M4
else'   = [i:=i+1]: loop'
```

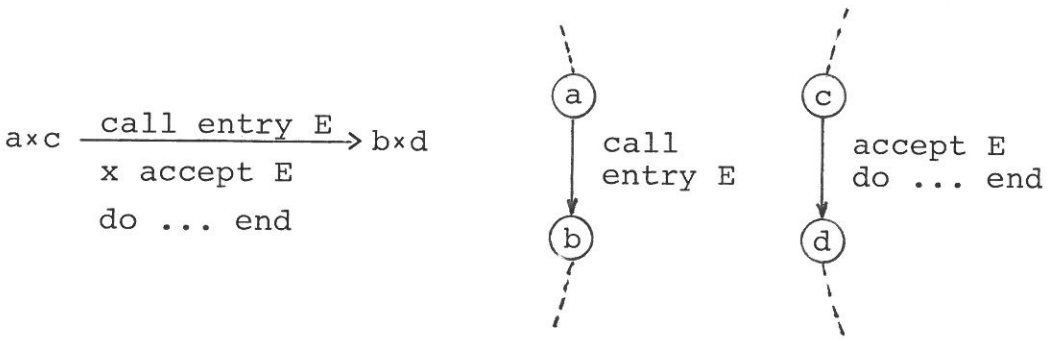
□

## #3 Parallel Programs

The essential difference between sequential and parallel programs is that only one control point in a sequential program can be active at any given time during a computation whereas a computation of a parallel program can have many active control points. How can we describe how the computation steps of a parallel program make some control points active and passify others? An appropriate mechanism is to use products for parallelism in the same way that we have been using sums for non-determinism. The difference between × and + has been the cause of much confusion when graphs have been used in other areas of computer science (× not + is appropriate for data flow graphs, database dependencies, and attribute grammar dependencies), but the difference has also been useful (AND-OR graphs, alternating Turing Machines).

Example

In ADA parallel tasks are synchronized by a rendezvous




If a and b are active, then they can be passified and both c and d made active; if control point c is passive, then the active control points a must remain active until c becomes active.
□

We want models, which allow both products and sums, but keep the simplicity of graph models. The appropriate choice seems to be diagrams;

- a set C of labelled configurations,
- a set M of moves, each of which is a label-source-sink triple;
- each source and sink of a move is built from C, using + and ×.

In the diagram model of a program configurations correspond to control points and moves correspond to combinations of program statements that must be executed concurrently.

Example

The procedure SEARCH has the diagram model with 5 moves

$$\text{begin} \xrightarrow{i:=1} \text{loop}$$

$$\text{loop} \xrightarrow{A(i)=\text{target},\ i=N} \text{then}_1 + \text{then}_2 + \text{else}$$

$$\text{then}_1 \xrightarrow{\text{return}(i)} \text{end}$$

$$\text{then}_2 \xrightarrow{\text{return}(0)} \text{end}$$

$$\text{else} \xrightarrow{i:=i+1} \text{loop}$$

and the configuration set (begin, loop, $\text{then}_1$, $\text{then}_2$, else).
□

Every flow chart is a diagram, because nodes correspond to configurations and frames correspond to moves; but the diagrams for flowcharts are special because move sources are configurations and move sinks are sums. Every graph is a diagram, because vertices correspond to configurations and edges correspond to moves; but the diagrams for graphs are special because move sources and sinks are configurations. Every equation set is a diagram because variables correspond to configurations and sums of terms correspond to moves; but diagrams for equation sets are special because move sinks are configurations and move sources are sums.

Once we have the diagram model of a program P, any question about P can be answered by induction

- assign assertions to the sinks and sources of every move;
- prove that each move preserves the truth of these assertions.

This proof method is sound because computations of a parallel program correspond to families of moves in the diagram. Often but not always one can assume that computations of a parallel

program correspond to sequences of diagram moves, not just families (multisets). If one makes this assumption for a diagram where all moves have configurations as sources and sinks, then diagram induction is just graph induction. When the source or sink of a diagram move is a configuration, the corresponding assertion is a "local invariant"; when the source or sink of a move is a sum or product, the corresponding assertion is a "global invariant". Global invariants play a prominent role in the literature on verifying parallel programs.

Diagrams can handle recursion because they usually have canonical solutions; such solutions are given by the algorithm

(Step1') assign assertions to the sources of all moves;
(Step2') calculate the assertion for the sinks of all moves;
(Step3') modify the assertions for move sources;
(Step4') if progress, then go to Step2'.

For programs in languages with strange semantics, the "solution" given by this algorithm may not be canonical because the algorithm makes implicit assumptions about simultaneous moves. Whether or not the algorithm gives canonical solutions, we still need a substitution combinator for making new diagrams from old. When presenting a diagram, one can use moves whose labels name diagrams. If the labelled move matches the named diagram, then the labelled move can be replaced by the set of matching moves. Clearly we must explain what we mean by matching. In a diagram $\mathcal{D}$ one can have I-configurations (O-configurations), which do not occur in the sink (source) of a $\mathcal{D}$-move. A move $m_0$ with label $\mathcal{D}$ <u>matches</u> a diagram $\mathcal{D}$ if and only if

- the number of configurations in the source of $m_0$ = the number of I-configurations in $\mathcal{D}$;
- the number of configurations in the sink of $m_0$ = the number of O-configurations in $\mathcal{D}$.

One gets the corresponding set of matching moves from $\mathcal{D}$ by

- replacing I-configurations by $m_0$-source-configurations;
- replacing O-configurations by $m_0$-sink-configurations;
- replacing all other configurations by new configurations.

We get precisely one set of matching moves if we assume an order on the set of all possible configurations.

Example

Earlier we gave the flowchart, graph and equation set models for a program MAIN and an internal block INIT. The corresponding diagrams are

$M1 \xrightarrow{INIT} M2$ $\qquad$ $I1 \xrightarrow{i:=1} I2$

$M2 \xrightarrow{Read(target)} M3$ $\qquad$ $I2 \xrightarrow{i=N} I5 + I3$

$M3 \xrightarrow{SEARCH} M4$ $\qquad$ $I3 \xrightarrow{A(i):=i} I4$

$M4 \xrightarrow{Write(i)} M5$ $\qquad$ $I4 \xrightarrow{i:=i+1} I2$

Using the substitution combinator and the diagram for SEARCH gives

$MI \xrightarrow{i:=1} I2'$ $\qquad$ $I2' \xrightarrow{i=N} M2 + I3'$

$M2 \xrightarrow{Read(target)} M3$ $\qquad$ $I3' \xrightarrow{A(i):=i} I4'$

$\qquad\qquad\qquad\qquad$ $I4' \xrightarrow{i:=i+1} I2'$

$M3 \xrightarrow{i:=1} loop'$ $\qquad$ $loop' \xrightarrow{A(i)=target,i=N} then_1' + then_2' + else$

$M4 \xrightarrow{Write(i)} M5$ $\qquad$ $then_1' \xrightarrow{return(i)} M4$

$\qquad\qquad\qquad\qquad$ $then_2' \xrightarrow{return(0)} M4$

$\qquad\qquad\qquad\qquad$ $else' \xrightarrow{i:=i+1} loop'$

□

## #4 Parallel Processes

In this section we show how diagram models capture the essence of two other popular formalisms for expressing parallelism: Petri nets and Milner agents. Among the many varieties of Petri nets [Pe] let us choose triples (P,T,R) where

- P is a set of labelled places;
- T is a set of labelled transitions;
- P and T are disjoint;
- R is a subset of $(P \times T) \cup (T \times P)$.

In a net model of a program places correspond to control points and transitions correspond to statements. There is a simple algorithm for converting a net to a diagram

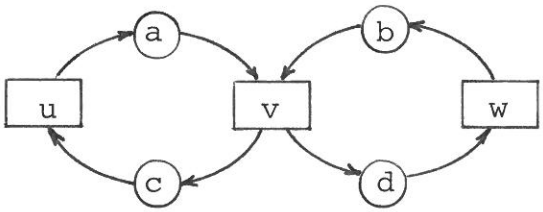
- places become configurations;
- transitions become moves;
- the source of the move for a transition t is the product of the configurations p such that $\langle p,t \rangle \in R$;
- the sink of the move for a transition t is the product of the configurations p such that $(t,p) \in R$.

Example

Consider the net: $P = (a,b,c,d)$

$T = (u,v,w)$

$R = ((a,v),(b,v),(c,u),(d,w),$
$(v,c),(v,d),(u,a),(w,b))$




The corresponding diagram has the moves: $c \xrightarrow{u} a$

$a \times b \xrightarrow{v} c \times d$

$d \xrightarrow{w} b$

□

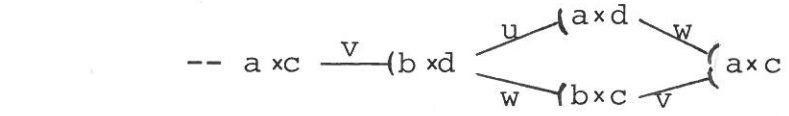
Although we have not used sums in our description of the algorithm for converting nets to diagrams, we should mention the connection between + and the net notion of conflict [Hi]. This connection is related to the controversy between "true parallelism", synchronization, and interleaving. For us this controversy is about the semantics of diagrams, and it does not affect the truth of

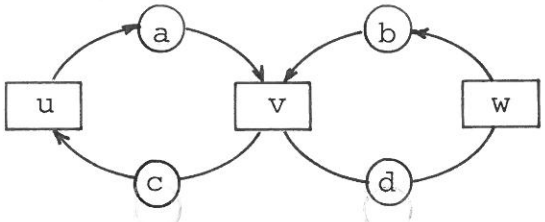
(*) Petri nets are particular kinds of diagrams.

Let us look briefly at three varieties of diagram semantics. A "true parallelism" semantics gives the meaning of a diagram as an event system [NeW,Wi], a set of move instances with a partial order "happens before". An interleaving semantics gives the meaning of a diagram as a set of move sequences, while a synchronizing semantics gives the meaning as a set of move product sequences.

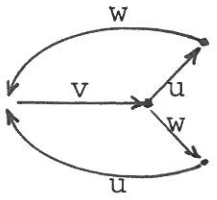
Example

"True Parallelism" ( )

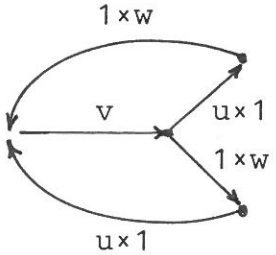



"Interleaving" is given by labels on paths in




"Synchronizing" is given by labels on paths in




□

From this example one might get the impression that the synchronization semantics is artificial, but this is not so: synchronization semantics is firmly based on category theory, and category theory rivals set theory as the foundation of mathematics. The category of a diagram has

- sums and products of configurations as objects;
- object identities, moves and their compositions as morphisms.

In the synchronization semantics the computation of a diagram corresponds to the morphisms in this category. Synchronization semantics is a generalization of [Go], where the paths in a graph also correspond to morphisms in a category and the semantics of a programming language provides a functor from this category to either partial functions or binary relations.

For programming languages the distinction between "true parallelism", "synchronization", and "interleaving" is seldom relevant because the meaning of programs does not depend on processor speeds. The speed of processors determines

- which move sequence in the set given to a program by an interleaving semantics actually occurs;
- which of the partially ordered sets of move instances given to a program by a true-parallelism semantics actually occurs;

but it should not determine the set of possible results.

Let us look at another popular formalism for parallelism, Milner agents [Mi1, Mi2]. The underlying idea is simple: any particular agent has a choice of actions it can perform, when an agent performs an action it changes into another agent. This idea gives a useful methodology for designing parallel programs [Ma]:

- divide the problem into cooperating sequential tasks;
- for each of these tasks make a graph with agents as vertex labels and actions as edge labels;
- capture task cooperation by synchronizing the actions.

In [Mi2] we find five powerful ways of building new agents from old:

| | |
|---|---|
| (SEQUENCE) | if a is an agent and $\alpha$ is an action, then $\alpha : a$ is an agent; |
| (SUMMATION) | if $(a_i)$ is a family of agents, indexed by $i \in I$, then $\Sigma_{i \in I} a_i$ is an agent; |
| (PRODUCT) | if $a_1$ and $a_2$ are agents, then $a_1 \times a_2$ is an agent; |
| (RESTRICTION) | if a is an agent and A is a set of actions, then $a \mid A$ is an agent; |
| (RECURSION) | if $(E_i)$ is a family of agent equations, indexed by $i \in I$, then $\underline{fix}_j (E_2)$ is an agent for each $j \in I$. |

In [Mi2] these agent construction methods are defined precisely, and we find proof rules that give a descendant graph for every agent. The vertices of this graph correspond to agents and there is an edge from $a_1$ to $a_2$ labelled $\alpha$ if and only if agent $a_1$ can perform action $\alpha$ and change into agent $a_2$.

Like any other graph models a descendant graph can be regarded as a diagram in which all moves have configurations as sink and source. For most parallel processes, that occur in current practice, the configurations (agents) in their descendant graphs are products and sums of agents in some class $\acute{A}$. If this is so, one gets a more natural diagram model by taking $\acute{A}$ as the set of configurations. In the particular case when only products are used, we have the much studied Cooperating Sequential Processes [Ho2].

Example

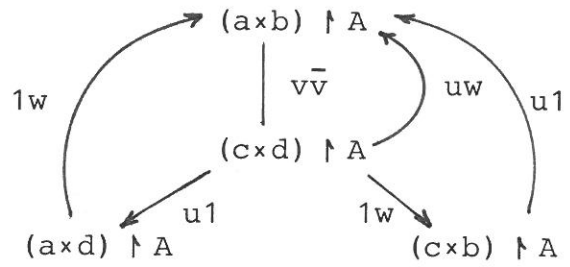
The family $(E_i)$ of agent equations

$$a = v{:}c, \quad c = u{:}a + 1{:}c, \quad b = \bar{v}{:}d, \quad d = w{:}b + 1{:}d$$

gives four agents

$$a = fix_1(E_i), \quad c = fix_2(E_i), \quad b = fix_3(E_i), \quad d = fix_4(E_i)$$

The product agent $(a\times b) \mid (v\bar{v}, uw, u1, 1w)$ has the descendant graph




where $A = (v\bar{v}, uw, v1, 1w)$.

The appropriate diagram model for this agent has the moves

$$a\times b \xrightarrow{v\bar{v}} c\times d, \quad c\times d \xrightarrow{u1} a\times d, \quad a\times d \xrightarrow{1w} a\times b$$

$$c\times d \xrightarrow{uw} a\times b, \quad c\times d \xrightarrow{1w} c\times b, \quad c\times b \xrightarrow{u1} a\times b$$

Because the sources and sinks of these moves are products, we have an example of sequential processes, (a,c) and (b,d), that cooperate with one another.

□

Some programming languages are restricted, from the text of a parallel program one can derive a bound on the number of processes that can be active. Other programming languages are freer, there is no way of bounding the number of tasks that can be active when an ADA program is executed - just as there is no way of bounding the number of incarnations of a procedure in a sequential language that allows recursion. How is this distinction reflected in diagram models? In a restricted language there is just one configuration for each program control point so the set

of configurations is finite; in a freer language this may still be true but the configurations must name the process, which owns the corresponding control point in the program. A more exact description of a suitable convention is

- the label of a configuration specifies the process which owns the configuration;
- if a configuration in the source of a move m is owned by a process P, that owns no configuration in the sink of m, them m destroys P;
- if a configuration in the sink of a move m is owned by a process P, that owns no configuration in the source of m, then m creates P.

Example

Suppose we add the moves

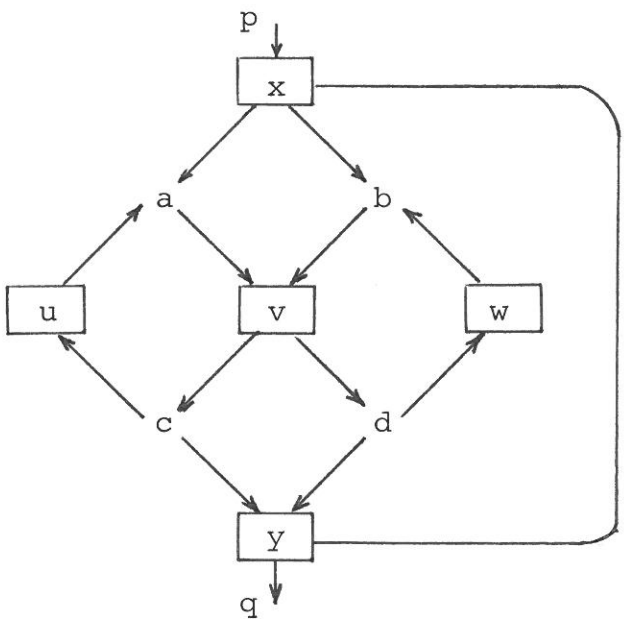
$$P \xrightarrow{x} a \times b, \quad c \times d \xrightarrow{y} q \times p$$

to the diagram in the last example. This corresponds to the agent $fix_1(E_j) \mid B$ where B is $(v\bar{v}, uw, u1, 1w, y\bar{y})$ and $E_j$ is the equation set

$$p = x{:}\ a\ b, \quad a = v{:}\ c, \quad b = \bar{v}{:}\ d,$$

$$c = u{:}\ a{+}1{:}\ c{+}y{:}\ q, \quad d = w{:}\ b{+}1{:}\ d{+}\bar{y}{:}\ q$$

The corresponding Petri net is



Since the sources and sinks of the new moves are products of configurations, the diagram still models cooperating sequential processes - but where are the processes? One answer is:

configurations (p,a,c) are owned by process P
configurations (b,d,q) are owned by process Q
x-moves create an incarnation of process Q

and incarnations of processes are never destroyed in this diagram. Because one can have many incarnations of process Q, product configurations like:

$$q \times q \times a \times b$$

are relevant when the appropriate semantics gives this diagram a meaning.
□

This example shows that diagrams can model not only the aspects of parallelism captured by Petri nets and Milner agents, but also the creation and destruction of parallel processes. In earlier sections we showed that diagram models can replace flowchart, graph and equation set models. There is a trend towards mixing models when giving the semantics of complex programming languages [JK], but diagram models seem to be sufficiently general to reverse this trend. This generality is illustrated by the table

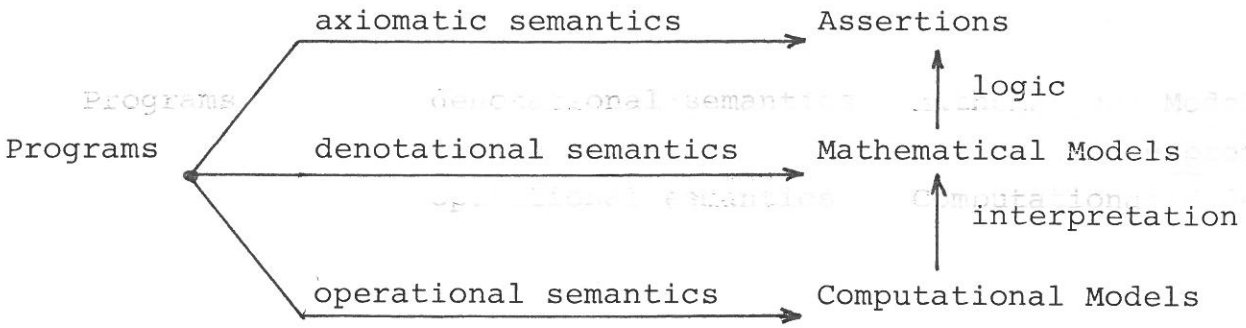
| | configuration | move | + | x |
|---|---|---|---|---|
| programs | control point | statement | non-determinism | parallelism |
| flowcharts | node | frame | non-determinism | NO |
| graphs | vertex | edge | non-determinism | NO |
| equation sets | variable | action | non-determinism | NO |
| Petri nets | place | transition | conflict | parallelism |
| Milner agents | agent | action | non-determinism | parallelism |

## #5 Semantics

At the outset we agreed that one understands a program if one can answer any relevant question about the program, but we have deliberately not specified what questions about a program P are relevant. If P is deterministic, questions like (1) does P give result r when started with data d? (2) does P stop, when started with data d? are relevant; if P is non-deterministic, questions like (3) does P always give result r, when started from data d? (4) do all fair computations of P from d stop? may become relevant. The notion of an "answer to a relevant question" must be formulated carefully, if one is not to run afoul of undecidable questions like the halting problem. Assume that relevant questions have a number of assertions as possible answers. One can answer a relevant question if there is a possible answer A such that

A is true ⇒ A can be proved formally.

With this requirement that answers to relevant questions are "semicomputable" assertions, we can present a view of programming language semantics:




Example

Each of the relevant questions, (1) to (4), has two possible answers: YES and NO. One can answer (1) or (2) because one can give a finite computation when the possible answer YES is true. Whether one can answer (3) or (4) depends on the semantics of the programming language:

- if the answer to (4) is NO, there is an infinite unfair computation of P from d, but one may not be able to prove this formally;
- if the answer to (3) is NO, there is either an infinite computation of P from d or a finite computation that gives a result different from r, but one may not be able to prove this formally. If the semantics of the language is such that the possible answer NO can be proved when true, then (3) and (4) are relevant questions, otherwise not.

□

In this paper we have described four kinds of computational models: flowcharts, graphs, equation sets and diagrams. These can be interpreted as mathematical models in many ways, but we shall only look at functional interpretations. An interpretation of a flowchart is given by assigning a function to each frame, then composing functions along paths; an interpretation of a graph is given by assigning a function to each edge, then composing functions along paths; an interpretation of an equation set is given by assigning functions to actions, then solving the equations; an interpretation of a diagram is given by assigning functions to moves, then composing functions along move sequences.

Let us turn to the "logic connection" between mathematical models and assertions. Usually it is easy

(Con) to decide if a finite set of assertions is true of some program in the language;

($\vdash$) to decide if a given assertion is a logical consequence of a given finite set of assertions.

These two requirements, (Con) and ($\vdash$), are precisely what is required to make a programming languages into a domain in Scott's new version of domain theory [Sc].

Example

Suppose the possible answers to questions about a program in a deterministic language are assertions of the form "P gives result r when started from data d". A finite set of such assertions can be true of some program if and only if the set does not contain two assertions

P gives result $r_1$ when started from data d
P gives result $r_2$ when started from data d

with different $r_1$ and $r_2$. The only consequences of a finite set of assertions are the assertions in the set. The deterministic programming language gives the domain of partial recursive functions from the data domain to the result domain.
□

Many theorists maintain that programming languages are just complicated examples of abstract data types (≡ domains). The aim of the semantics of a language is to define a particular element of this data type as the meaning of a program in the language. The syntax of a language usually helps the specification of the meaning of a program P, because this meaning is determined by the meanings of smaller and smaller syntactic fragments of P. Not only programs form a domain; every syntactic category (non-terminal symbol) forms a domain, and there is a connection between these domains for each syntactic rule. The role of the program models, discussed in this paper, is to reduce semantics to the specification of models for program fragments, and this is enough because assertions about program fragments are logically equivalent to assertions about the corresponding models.

References

[ADJ] Initial algebra semantics and continuous algebras. J. ACM 24 (1977).

[CKS] A.K. Chandra, D. Kozen, L. Stockmeyer: Alternatives. J. ACM 28 (1981).

[CFZ] B. Courcelle, P. Franchi-Zanetacci: Attribute grammars and recursive program schemes, Th. CS 17 (1981) 163-191, 235-257.

[D] E.W. Dijkstra: A discipline of programming. Prentice-Hall, 1976.

[F] R.W. Floyd: Assigning meanings to programs. Proc. Symp. Appl. Math. 19 (1967) 19-32.

[Go] J.A. Goguen: On homomorphisms, correctness, termination, unfoldments, and equivalence of flow diagram programs, JCSS 8 (1974) 333-365.

[Gr] Graph grammars and their application to computer science and biology. LNCS 73, Springer 1973.

[Gu] I. Guessarian: Algebraic semantics. LNCS 99, Springer 1979.

[Hi] W. Hinderer: Transfer of graph constructs in Goguen's paper to net constructs. In: Informatik Fachberichte 52, Springer 1982.

[Ho1] C.A.R. Hoare: An axiomatic basis for computer programming. CACM 12 (1969) 576-583.

[Ho2] C.A.R. Hoare: Communicating Sequential Processes. CACM 21 (1978) 666-677.

[Hu] G. Huet: Confluent reductions: abstract properties and applications to term rewriting systems. JACM 27 (1980) 797-821.

[JK] K. Jensen, M. Kyng: EPSILON, a system description language. DAIMI PB-150, Aarhus 1982.

[K] R.M. Keller: Formal verification of parallel programs. CACM 19 (1976) 371-384.

[Ma] B. Mayoh: Parallelism in ADA: program design and meaning. In: LNCS 83 (1980) 256-268.

[Mi1] R. Milner: A calculus of communicating systems. LNCS 92, Springer 1980.

[Mi2] R. Milner: Calculi for synchrony and asynchrony. Edinburgh report CSR-104-82.

[NPW] M. Nielsen, G. Plotkin, G. Winskel: Petri nets, event structures, and domains. TCS 13 (1981) 85-99.

[Pa] P. Padawitz: Graph grammars and operational semantics. CS 19 (1982) 117-141.

[Pe] C.A. Petri: Concurrency. In: LNCS 84, Springer 1980.

[Pl] G. Plotkin: A structural approach to operational semantics. DAIMI FN-19, Aarhus 1982.

[Pn] A. Pnueli: The temporal semantics of concurrent programs. TCS 13 (1979) 45-60.

[Pr] V. Pratt: Semantical considerations of Floyd-Hoare logic. Proc. 17 FOCS, IEEE, 1976.

[Sa] A. Salwicki: Formalized Algorithmic Languages. Bull. Acad. Pol. Sci., Ser. Math. Astr. Phys. 18 (1970) 227-232.

[Sc] D. Scott: Domains for denotational semantics. In: LNCS 140, Springer 1982.

[St] J. Stoy: Denotational semantics: the Scott-Strachey approach to programming language theory. MIT Press 1977.

[Wi] G. Winskel: Event structure semantics for CCS and related languages. In: LNCS 140, Springer 1982.