STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-142

COMPUTER SCIENCE DEPARTMENT
REPORT NO. CS-205

AN ALGEBRAIC DEFINITION OF SIMULATION BETWEEN PROGRAMS

BY

ROBIN MILNER

FEBRUARY 1971

COMPUTER SCIENCE DEPARTMENT

STANFORD UNIVERSITY

STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO NO. AIM-142

FEBRUARY 1971

COMPUTER SCIENCE DEPARTMENT
REPORT NO. CS205

AN ALGEBRAIC DEFINITION OF SIMULATION BETWEEN PROGRAMS

by: Robin Milner

ABSTRACT: A simulation relation between programs is defined which is quasi-ordering. Mutual simulation is then an equivalence relation, and by dividing out by it we abstract from a program such details as how the sequencing is controlled and how data is represented. The equivalence classes are approximations to the algorithms which are realized, or expressed, by their member programs.

A technique is given and illustrated for proving simulation and equivalence of programs; there is an analogy with Floyd's technique for proving correctness of programs. Finally, necessary and sufficient conditions for simulation are given.

DESCRIPTIVE TERMS: Simulation, weak homomorphism, algorithm, program correctness, program equivalence.

This research was supported mainly by the Science Research Council, Great Britain and in part by the Advanced Research Projects Agency of the Department of Defense (SD-183) U.S.A.

1. INTRODUCTION

One aim of this paper is to make precise a sense in'which two programs may be said to be realizations of the same algorithm. We can say loosely that for this to be true it is sufficient though perhaps not necessary that the programs do the same 'important' computations in the same sequence, even though they differ in other ways: for example (1) We may disregard other computations perhaps different in the two programs, which are 'unimportant' in the sense that they are only concerned with controlling the 'important' ones, (2) The data may flow differently through the variables or registers, (3) The data may be differently represented in the two programs. The program pairs in Figures 1, 2, which are studied in detail in Section 4, illustrate points (1), (3) respectively; a trivial illustration of (2) is the following pair of programs:

```
read x,y          read x,y
x := x + y        y := x + y
print x           print y
```

Although the above prescription is vague, we give a relation of simulation between programs which may fairly be said to match it. The relation turns out to be transitive and reflexive but not always symmetric; however mutual simulation is an equivalence relation, and it is the equivalence classes under this relation which may be regarded as algorithms - at least this is an approximation to a definition of algorithm.

We show also that there is a practical technique for proving simulation in interesting cases - though unfortunately simulation between programs handling the integers, for example, is not a decidable (or even

partially decidable) relation. Under a simple restriction simulation ensures the equivalence (as partial functions) of the programs, so this is also a technique for proving equivalence; however in general equivalent programs will not satisfy the simulation relation.

I also claim that in order to prove by Floyd's [1] method the correctness of a program A, in a case where data is represented unnaturally, perhaps for efficiency's sake, the easiest and most lucid approach is rather close to first designing a program B which is simulated by program A and which represents the data naturally, and then proving B correct. This was in fact the original motivation for studying simulation, and is discussed in more detail in Milner [2], which contains a first attempt at the definition of simulation. The sequel [3] generalizes the definition and the current paper is a synthesis of the two, and may be read independently.

2. NOTATION

We denote relational composition by juxtaposition; if $R \subseteq A \times B$, $S \subseteq B \times C$ then $RS = \{\langle a,c\rangle \mid \exists b.\langle a,b\rangle \in R, \langle b,c\rangle \in S\}$. The inverse of R is $R^{-1} = \{\langle b,a\rangle \mid \langle a,b\rangle \in R\}$. We intentionally confuse a (partial) function $F: A \to B$ with the relation $F = \{\langle a,b\rangle \mid b=F(a)\}$. R induces a function $Im_R: 2^A \to 2^B$; for $S \subseteq A$, $Im_R(S) = \{b \mid \exists a \in S.\langle a,b\rangle \in R\}$. For a function F we sometimes write $F(S)$ for $Im_F(S)$. The domain of $R \subseteq A \times B$ is dom R $= Im_R(A)$, and the range of R is ran R $= m_{R\text{-}l(B)=}$ For any set A, $I_A \subseteq A \times A$ is the identity relation on A.

3. PROGRAMS AND SIMULATION

We first introduce a definition of program which enables simulation properties to be stated and proved succinctly.

Definition. A program $\mathcal{A}$ is a quadruple $(D_{in}, D_{comp}, D_{out}, F)$ where $D_{in}, D_{comp}, D_{out}$ are disjoint domains and $F: D \to D$ is a total function $(D = D_{in} \cup D_{comp} \cup D_{out})$ with restrictions

(i) $F(D_{in} \cup D_{comp}) \subseteq D_{comp} \cup D_{out}$

(ii) The restriction of F to D_{out} is the identity $I_{D_{out}}$

We call $D_{in}, D_{comp}, D_{out}$ the input, computation and output domains of $\mathcal{A}$.

Conditions (i) and (ii) ensure that starting with'a member of D_{in} and applying F repeatedly we get either an infinite sequence in D_{comp}, or a finite sequence in D_{comp} followed by a single repeated member of D_{out}. We have (ii) merely to keep F total, which the theory requires.

Why must the domains be disjoint? What about a program which inputs an integer and outputs an integer? Here one might argue that $D_{in} = D_{out}$ = [integers]; but we get into no trouble having two formally disjoint domains with for example an injection or a bijection between them. In fact, in practice we can distinguish between an input object and an output object of a program; for example they occur on different media, or at different spatial locations. We are concerned with a level of abstraction (i.e. abstraction from real computers operating on physical data symbols) lower than that in which a program is considered as for example a function from integers to integers.

Definition. A computation sequence of $\mathcal{A}$ is a sequence $\{d_i \mid i \geq 0\}$ where $d_0 \epsilon D_{in}$, $d_{i+1} = F(d_i)$, $i \geq 0$, and either $d_i \epsilon D_{comp}$, $i > 0$ or for some k $d_i \epsilon D_{comp}$, $0 < i < k$ and $d_i = d_k \epsilon D_{out}$, $i \geq k$.

Definition. A program α determines its associated partial function

$$\hat{\alpha} : D_{in} \rightarrow D_{out}$$

in an obvious way.

Often we would have $D_{comp} = N \times E$, where E is the set of possible state-vector values, and for non-recursive (flowchart) programs N is the finite set of nodes of the flowchart while for recursive programs N is the infinite set of possible states of a pushdown store.

Before dealing with simulation, we state without proof some theorems concerning correctness and termination of programs. Theorem 3.1 embodies Floyd's [1] method of proving partial correctness of programs. There is also a correspondence with Manna's work - for example in [4]; on Theorems 3.1 and 3.2 correspond to Theorems 1 and 2 of that paper. However, Manna is concerned with the representability of verifications (as defined below) in first order predicate calculus; we perhaps gain in succinctness by stating results algebraically and ignoring the question of representability.

Hence forward we assume that the suffix 'in' to a symbol denoting a set implies inclusion in D_{in}. Similarly for 'comp', 'out'.

Definitions. α is partially correct w.r.t. S_{in}, S_{out} if $\hat{\alpha}(S_{in}) \subseteq S_{out}$.

α is totally correct w.r.t. S_{in}, S_{out} if in addition $\hat{\alpha}$ is total on S_{in}.

S is a verification of α or S verifies α, if $S \subseteq D$ and $F(S) \subseteq S$.

Theorem 3.1

Given S_{in}, S_{out}

[α partially correct w.r.t. S_{in}, S_{out}] $\Leftrightarrow$

[There is a verification $S_{in} \cup S_{comp} \cup S_{out}$ of α] □

Theorem 3.2

Given S_{in}, S_{out}

[σ totally correct w.r.t. S_{in}, S_{out}] $\Leftrightarrow$

[For every verification $S'_{in} \cup S'_{comp} \cup S'_{out}$ of σ,

$S_{in} \cap S' \neq \emptyset \Rightarrow S_{out} \cap S'_{out} \neq \emptyset$] $\square$

Corollary. (Set $S_{out} = D_{out}$)

[σ total on S_{in}] $\Leftrightarrow$

[For every verification $S'_{in} \cup S'_{comp} \cup S'_{out}$ of σ, $S_{in} \cap S'_{in} \neq \emptyset \Rightarrow S'_{out} \neq \emptyset$] $\square$

Now assume two programs, $\sigma = \langle D_{in}, D_{comp}, D_{out}, F \rangle$

and $\sigma' = \langle D'_{in}, D'_{comp}, D'_{out}, F' \rangle$.

Definition. Let $R \subseteq D \times D'$. Then R is a weak simulation of σ by σ' if

(i) $R \subseteq D_{in} \times D'_{in} \cup D_{comp} \times D'_{comp} \cup D_{out} \times D'_{out}$

(ii) $R\, F' \subseteq F\, R$.

Condition (ii) simply states that R is a weak homomorphism between the algebraic structures $\langle D, F \rangle, \langle D', F' \rangle$. This concept is used in automata theory to define the notion of covering - see for example Ginzburg [7, p. 98].

Now denote $R \cap (D_{in} \times D'_{in})$ by R_{in}, and R_{comp}, R_{out} similarly, so that $R = R_{in} \cup R_{comp} \cup R_{out}$ and these parts are disjoint.

Theorem 3.3.

If R is a weak simulation of σ by σ' then

(i) $R_{in} \hat{\sigma}' \subseteq \hat{\sigma}\, R_{out}$

(ii) R^{-1} is a weak simulation of σ' by σ

(iii) $R^{-1}_{in}\, \hat{\sigma} \subseteq \hat{\sigma}'\, R^{-1}_{out}$

Proof (i) The condition $R\,F' \subseteq F\,R$ may be restated

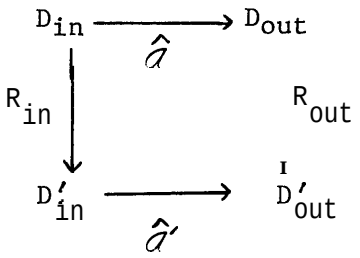
$$\forall d,d'.\ \langle d,d'\rangle \in R \Rightarrow \langle F(d),F'(d')\rangle \in R \qquad (*)$$

Now suppose $\langle d_o,d'_k\rangle \in R_{in}\ \hat{a}'$. Then for some d'_o, $\langle d_o,d'_o\rangle \in R_{in}$ and $\hat{a}'(d'_o) = d'_k$, so there is a computation sequence $d'_0,d'_1 \ldots\ldots,d'_k,\ldots\ldots$ of a'. Now consider the computation sequence $d_o,d_1,\ldots.d_k,\ldots.$ of a. We may prove by induction using (*) that $\langle d_i,d'_i\rangle \in R$, $i \geq 0$. Hence $d_k \in D_{out}$, $\langle d_o,d_k\rangle \in \hat{a}$, $\langle d_k,d'_k\rangle \in R_{out}$ and so $\langle d_o,d'_k\rangle \in \hat{a}\ R_{out}$.

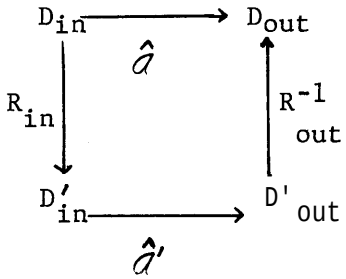
(ii) It is enough to show $R^{-1}\,F \subseteq F'\,R^{-1}$. But this follows easily from the fact that (*) is equivalent to $RF' \subseteq FR$.

(iii) Follows directly using (i) and (ii). □

Theorem 3.3(i) says that the diagram



semi-commutes (i.e. we have $\subseteq$ not $=$). If we wish to be able to use a' to do the job of a, we need more: we need the following to commute



i.e. we require $\hat{a} = R_{in}\ \hat{a}'\ R^{-1}_{out}$. Theorem 3.4 below shows that for this it is sufficient to require R to be a strong simulation of a by a', where

Definition. A weak simulation R of $\mathcal{A}$ by $\mathcal{A}'$ is a strong simulation if in addition R_{in}, R^{-1}_{out} are total and single valued.

Note that R^{-1} is not necessarily a strong simulation of $\mathcal{A}'$ by $\mathcal{A}$, so unlike weak simulation, strong simulation is not symmetric.

Theorem 3.4

If R is a strong simulation of $\mathcal{A}$ by $\mathcal{A}'$ then $\hat{\mathcal{A}} = R_{in}\,\hat{\mathcal{A}}'\,R^{-1}_{out}$.

Proof ($\supseteq$) Post multiply Theorem 3.3(i) by R^{-1}_{out} and use $R_{out}R^{-1}_{out} \subseteq I_{D_{out}}$ (R^{-1}_{out} single valued).

($\subseteq$) Premultiply Theorem 3(ii) by R_{in} and use $I_{D_{in}} \subseteq R_{in}R^{-1}_{in}$ (R_{in} total). □

(Note that in the above we did not use the totality of R^{-1}_{out}, nor the single valuedness of R_{in}).

Let us return to the discussion of algorithm in the introduction. If there is a strong simulation of $\mathcal{A}$ by $\mathcal{A}'$ we say $\mathcal{A}'$ strongly simulates $\mathcal{A}$, and it is easy to show that this is a transitive reflexive relation, i.e., a quasi-ordering. Mutual strong simulation is therefore an equivalence relation, and the equivalence classes may be thought of as algorithms, each of which is realized by its member programs. Moreover, if we divide out by this equivalence relation we obtain from the quasi-ordering of programs a partial ordering of algorithms.

It is worth noticing that there is always a weak simulation between any pair of programs - just take $R = \phi$ - so a similar definition of "$\mathcal{A}'$ weakly simulates $\mathcal{A}$" is vacuous.

We finish this section with two simple results which exhibit the close relationship between verifications and simulations.

Theorem 3.5

If R is a simulation of $\mathcal{A}$ by $\mathcal{A}'$ then

(i) dom R verifies $\mathcal{A}$

(ii) ran R verifies $\mathcal{A}'$.

Proof In view of Theorem 3.3(ii) we only prove (i). Clearly $\mathrm{domR} \subseteq D$, and we only need show $F(\mathrm{domR}) \subseteq \mathrm{domR}$, i.e. $\forall d.\ d \in \mathrm{domR} \Rightarrow F(d) \in \mathrm{domR}$.

But $d \in \mathrm{domR} \Rightarrow \exists d'.\ \langle d,d' \rangle \in R$

$\Rightarrow \exists d'.\ \langle F(d),F'(d') \rangle \in R$

$\Rightarrow F(d) \in \mathrm{domR}$. □

This theorem says that simulation of $\mathcal{A}$ by $\mathcal{A}'$ implies the partial correctness of $\mathcal{A}$ w.r.t. $\mathrm{domR}_{in}, \mathrm{domR}_{out}$. However, normally we are interested in partial correctness w.r.t. an S_{in}, S_{out} where S_{out} is much smaller than $\mathrm{dom}\ R_{out}$; for example if R_{out} is total then $\mathrm{dom}\ R_{out} = D_{out}$, and $\mathcal{A}$ is always partially correct w.r.t $S_{in\ out}$ when $S_{out} = D_{out}$.

Theorem 3.6

If S verifies $\mathcal{A}$ and R is a simulation of $\mathcal{A}$ by $\mathcal{A}'$ then $\mathrm{Im}_R(S)$ verifies $\mathcal{A}'$.

Proof We require $F'(\mathrm{Im}_R(S)) \subseteq \mathrm{Im}_R(S)$.

But $F'(\mathrm{Im}_R(S)) = \mathrm{Im}_{RF'}(S)$

$\subseteq \mathrm{Im}_{FR}(S)$ since $RF' \subseteq FR$

$= \mathrm{Im}_R(F(S))$

$\subseteq \mathrm{Im}_R(S)$ since $F(S) \subseteq S$. □

Thus in a precise sense a proof of partial correctness of $\mathcal{A}'$ may be factored into a proof of partial correctness of $\mathcal{A}$ together with a proof of simulation of $\mathcal{A}$ by $\mathcal{A}'$.

4. APPLICATION TO FLOWCHART PROGRAMS

In this section, we show how we may demonstrate a simulation between two programs in a manner which bears a close relation to Floyd's method for proving correctness of a single program. Of the two examples, the first has the same data representation but different control in the two programs.

Given a flowchart program with input domain D_{in}, state-vector domain E, output domain D_{out} and nodeset N, and given also an input function $f_{in}: D_{in} \rightarrow E$ and output function $f_{out}: E \rightarrow D_{out}$, it is a simple matter to formalize it as a program according to our definition, with $D_{comp} = N \times E$ and $F: D \rightarrow D$ defined in terms of f_{in}, f_{out} and the tests and assignments in the boxes. Alternatively, we may formalize it by selecting a subset $M \subseteq N$ so that every cycle in the flowchart contains a member of M (we call such an M a cycle-breaking set) and define D_{comp} instead as $M \times E$. The cycle breaking property ensures that $F : D \rightarrow D$ is again total.

Now suppose in $\mathcal{A}$ and $\mathcal{A}'$ we have $D_{comp} = M \times E$, $D'_{comp} = M' \times E'$. $\mathcal{A}$ and $\mathcal{A}'$ may have been obtained by the above formalization from flowchart programs, for example. If R is a simulation of $\mathcal{A}$ by $\mathcal{A}'$, we have $R_{comp} \subseteq (M \times E) \times (M' \times E')$, and to exhibit R_{comp} it is sufficient to exhibit $R_{mm'}$, for each $m \in M$, $m' \in M'$ where

$$R_{mm'} = \{\langle e, e' \rangle \mid \langle \langle m, e \rangle, \langle m', e' \rangle \rangle \in R_{comp}\}.$$

In the following two examples we exhibit the $R_{mm'}$, and also indicate how the proof of $RF' \subseteq FR$ would go.

Example 1. (See Figure 1). Assume that inputs to each program are

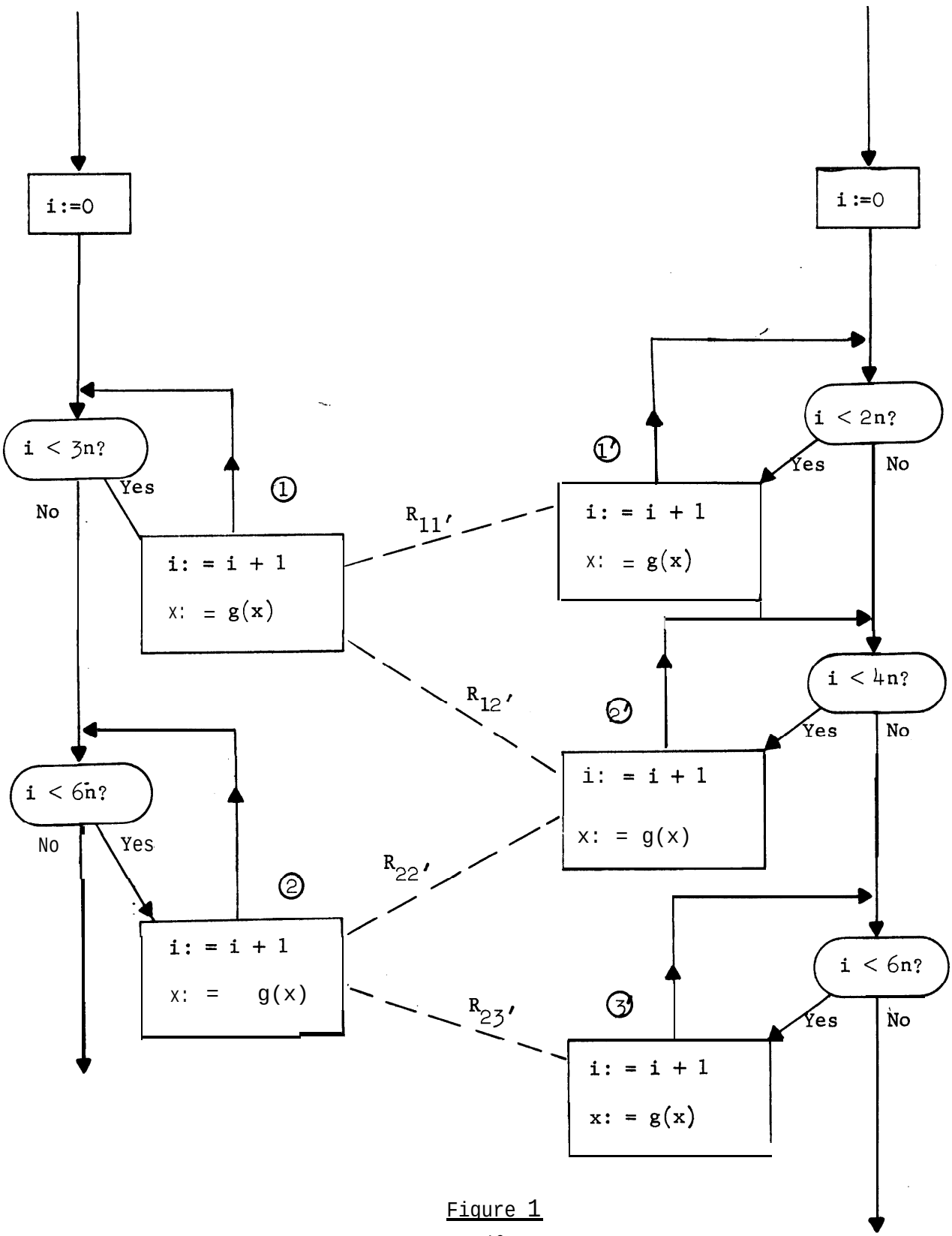

PROGRAM Q
PROGRAM Q'
i:=0
i < 3n?
Yes
No
①
i: = i + 1
x: = g(x)
i < 6n?
Yes
No
②
i: = i + 1
x: = g(x)
R11'
R12'
R22'
R23'
i:=0
i < 2n?
Yes
No
1'
i: = i + 1
x: = g(x)
i < 4n?
Yes
No
2'
i: = i + 1
x: = g(x)
i < 6n?
Yes
No
3'
i: = i + 1
x: = g(x)

Figure 1

pairs $\langle n,x\rangle$, state vectors are triples $\langle i,n,x\rangle$, and only x is output. The node-set $\{1,2\}$ has been chosen to formalize $\mathcal{A}$, and $\{1',2',3'\}$ to formalize $\mathcal{A}'$. So if $\mathcal{I},\mathcal{R}$ denote integers and reals we have

$$D_{in} = D'_{in} = \mathcal{I} \times \mathcal{R};$$
$$E = E' = \mathcal{I} \times \mathcal{I} \times \mathcal{R};\ D_{comp} = \{1,2\} \times E;\ D'_{comp} = \{1',2',3'\} \times E;$$
$$D_{out} = D'_{out} = \mathcal{R}$$

and F, the transition function for $\mathcal{A}$, is given by

$F(d)$ = **if** $d \in D_{in}$ **then** $\langle 1,\langle 0,n,x\rangle\rangle$ **where** $\langle n,x\rangle = d$

else if $d \in D_{comp}$ **then let** $\langle m,\langle i,n,x\rangle\rangle = d$;

if $i + 1 < 3n$ **then** $(m,(i + 1,n,g(x))\rangle$

else if $i + 1 < 6n$ **then** $\langle 2,\langle i + 1,n,g(x)\rangle\rangle$

else $g(x)$

else d

F' for $\mathcal{A}'$ is defined similarly.

We postulate a simulation R by giving R_{in}, R_{out} and $R_{mm'}$, for $\langle m,m'\rangle \in \{1,2\} \times \{1',2',3'\}$, as follows:

$$R_{in} = I_{Din};\ R_{out} = I_{Dout};\ R_{13'} = R_{21'} = \emptyset;$$
$$R_{11'} = \{\langle\langle i,n,x\rangle,\langle i,n,x\rangle\rangle \mid i < 2n\}$$
$$R_{12'} = \{\langle\langle i,n,x\rangle,\langle i,n,x\rangle\rangle \mid 2n \leq i < 3n\}$$
$$R_{22'} = \{\langle\langle i,n,x\rangle,\langle i,n,x\rangle\rangle \mid 3n \leq i < 4n)$$
$$R_{23'} = \{\langle\langle i,n,x\rangle,\langle i,n,x\rangle\rangle \mid 4n \leq i < 6n\}$$

For example, we may think of $R_{12'}$ as containing all state-vector pairs attained at the node-pair $\langle 1,2'\rangle$ when $\mathcal{A},\mathcal{A}'$ are obeyed synchronously starting from an input pair in R_{in}. However, it contains also many other

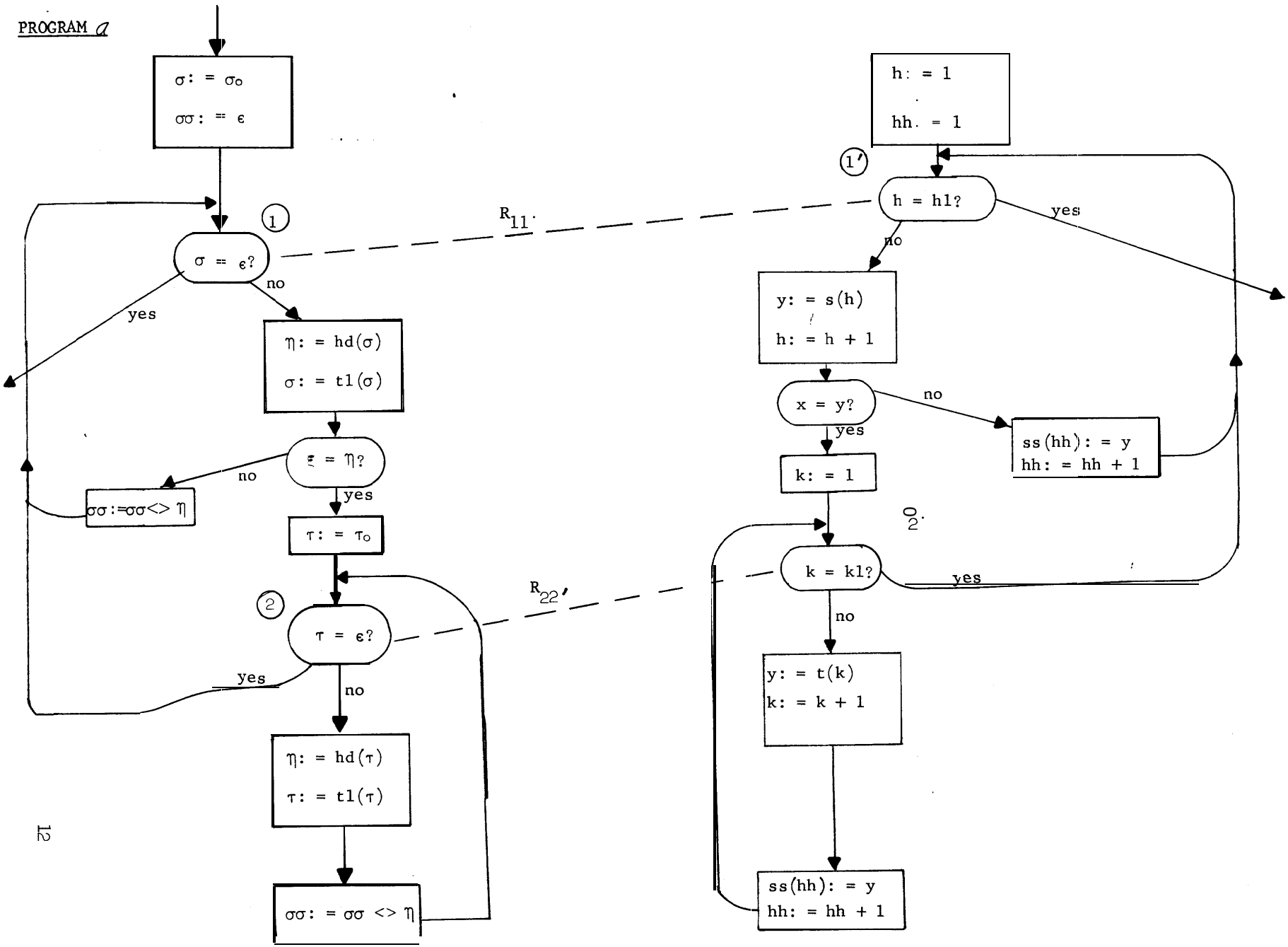

PROGRAM a
σ: = σo
σσ: = ε
1
σ = ε?
yes
no
η: = hd(σ)
σ: = tl(σ)
ξ = η?
no
yes
σσ:=σσ<> η
τ: = τo
2
τ = ε?
yes
no
η: = hd(τ)
τ: = tl(τ)
σσ: = σσ <> η
R11
R22'
h: = 1
hh. = 1
1'
h = hl?
yes
no
y: = s(h)
h: = h + 1
x = y?
no
yes
ss(hh): = y
hh: = hh + 1
k: = 1
2'
k = kl?
yes
no
y: = t(k)
k: = k + 1
ss(hh): = y
hh: = hh + 1

Figure 2

state-vector pairs (since there is no constraint on x in the definition of $R_{12'}$), and simulation will normally have this generous property. $R_{13'}$ is here taken as the empty set, because the node pair $\langle 1,3' \rangle$ is never reached.

To prove RF' $\subseteq$ FR we must show that for all d,d'

$$\langle d,d' \rangle \in R \Rightarrow \langle F(d),F'(d') \rangle \in R$$

and this may be done by cases

Case 1: $\langle d,d' \rangle \in R$

Case 2: $\langle d,d' \rangle \in R_{out}$

Case $3_{mm'}$: $\langle d,d' \rangle = \langle \langle m,e \rangle, \langle m',e' \rangle \rangle$ where $\langle e,e' \rangle \in R_{mm'}$

which is a fairly--routine matter using the definitions of F,F', and we leave it to the reader.

Now since R is a strong simulation, and indeed R_{in}, R_{out} are identities, Theorem 3.4 entitles us to conclude $\hat{a} = \hat{a}'$.

Example 2. (See Figure 2). This example illustrates simulation between two programs with different data representation. We describe this example in less detail, to save space. Each program is supposed to input a string σ, a character ξ and a string τ, and to output the result of substituting τ for ξ everywhere in σ. Thus if S is the alphabet of characters, $D_{in} = D'_{in} = S^* \times S \times S^*$ (where S^* is the set of strings over S) and $D_{out} = D'_{out} = S^*$. Program α handles strings and characters directly, using the functions hd,tl,$\langle\rangle$(concatenation) and the null string ϵ. The three inputs are to the program variables σ_0, ξ, τ_0 respectively, and output is from the variable $\sigma\sigma$. On the other hand, program α' represents each string as a segment of an integer-indexed character array; on input the two input strings are stored in arrays s,t (indexed from 1) their -lengths + 1 in integer variables hl, kl and the

character in x, and output is the string $ss(1),ss(2).....ss(hh-1)$.

The flowcharts are formalized as programs (in our sense) with node-sets $\{1,2\}, \{1',2'\}$, and we have $D_{comp} = \{1,2\}$ x (the set of possible values for the program variable vector of $\mathcal{A}$), and similarly for D'_{comp}. F and F', the transition functions, are easy but tedious to define. We now exhibit a simulation by giving R_{in}, R_{out} and the $R_{mm'}$ for $\langle m,m'\rangle \in \{1,2\} \times \{1',2'\}$, using an auxiliary function seq: arrays x integers x integers 3 strings defined by

$$seq(a,i,j) = \begin{cases} \text{The string } a(i),a(i+1),.....a(j-1) \text{ if } i \leq j \\ \text{Arbitrarily defined if } i > j. \end{cases}$$

$$R_{in} = I_{D_{in}}; R_{out} = I_{D_{out}}; R_{12'} = R_{21'} = \emptyset;$$

$$R_{11'} = \{\langle\langle\sigma_o,\sigma,\sigma\sigma,\xi,\eta,\tau_o,\tau\rangle,\langle s,h,hl,ss,hh,x,y,t,k,kl\rangle\rangle \mid P_1\};$$

$$R_{22'} = \{\langle\langle\sigma_o,\sigma,\sigma\sigma,\xi,\eta,\tau_o,\tau\rangle,\langle s,h,hl,ss,hh,x,y,t,k,kl\rangle\rangle \mid P_2\}$$

where $P_1 \equiv \sigma_o = seq(s,1,hl) \wedge \sigma = seq(s,h,hl)$ A $\tau_o = seq(t,1,kl) \wedge$

$\sigma\sigma = seq(ss,1,hh)$ A $\xi = x \wedge 1 \leq h \leq hl \wedge 1 \leq kl \wedge 1 \leq hh$

and $P_2 \equiv P_1$ A $\tau = seq(t,1,k)$ A $\eta = y$ A $1 \leq k \leq kl$.

Now as in Example 1 the proof of $RF' \subseteq FR$ must proceed by cases; it will use certain properties (or axioms) concerning the string handling functions, the array and integer handling functions and the function seq. We leave it to the reader again. Again, since R_{in}, R_{out} are identities we have proved that $\hat{\mathcal{A}} = \hat{\mathcal{A}'}$.

There are some interesting points about this example.

(1) It seems that program $\mathcal{A}$ is more natural than $\mathcal{A}'$, though this asymmetry was not present in Example 1. In fact, program $\mathcal{A}'$ is only a slight modification of part of a real program written for use rather than

as an example. In the process of proving α' correct using Floyd's technique, I found that the assertions associated with parts. of the program were most naturally expressed using the function seq, and that the terms appearing in these assertions were precisely those which are here related (in R_{11}, and R_{22},) to the variables of α. In fact, (this is discussed in more detail in [2]) the task of proving α' correct factored simply into two tasks - that of proving α correct (an easier task since α is more natural and closer to programmer's intuition) and that of proving the simulation. This 'factoring' was made precise by Theorem 3.6.

(2) Unlike in Example 1, the flowcharts here have identical shape, and it is meaningful (and even true!) to say that under identical inputs the programs follow the same path. In Example 1 such a statement would not be meaningful, but in Section 5 we show that a similar statement has meaning in cases more general then Example 2, and provides us with necessary and sufficient conditions 'for the existence of a simulation between two programs.

5. PARTITIONED SIMULATION

We now obtain necessary and sufficient conditions for the existence of simulation between two programs α and α'.

Definition. If J is any indexing set and $\pi_J = \{C_j \mid j \in J\}, \pi'_J = \{C'_j \mid j \in J\}$ are partitions of D_{comp}, D'_{comp} respectively, then $\langle \pi_J, \pi'_J \rangle$ is a partition pair for D_{comp}, D'_{comp} (Of course any two domains can have a partition pair, but we are only concerned with computation domains).

Definition. Computation sequences $\{d_i\}$ in α, $\{d'_i\}$ in α' agree for $\langle \pi_J, \pi'_J \rangle$ if $\forall i,j.\ d_i \in C_j \iff d'_i \in C'_j$.

Definition. A simulation R respects $\langle \pi_J, \pi'_J \rangle$ if $R_{comp} \subseteq \bigcup_{j \in J} (C_j \times C'_j)$

Theorem 5.1

(Weak Simulation Theorem). Given $R^*_{in} \subseteq D_{in} \times D'_{in}$ and a partition pair $\langle \pi_J, \pi'_J \rangle$ of D_{comp}, D'_{comp}, the following two statements are equivalent:

(a) Computation sequences $\{d_i\}$ of $\mathcal{A}$, (df) of $\mathcal{A}'$ for which $\langle d_o, d'_o \rangle \in R^*_{in}$ always agree for $\langle \pi_J, \pi'_J \rangle$.

(b) There is a weak simulation $R = R^*_{in} \cup R_{comp} \cup R_{out}$ of $\mathcal{A}$ by $\mathcal{A}'$ which respects $\langle \pi_J, \pi'_J \rangle$.

Proof (a) $\Rightarrow$ (b). It is enough to take

$R = \{\langle e, e' \rangle \mid$ There are computation sequences $\{d_i\}$ of a, $\{d'_i\}$ of $\mathcal{A}'$ for which $\langle d_o, d'_o \rangle \in R^*_{in}$ and for some k $e = d_k, e' = d'_k\}$.

(b) $\Rightarrow$ (a). Assume R. Take any computation sequences $\{d_i\}, \{d'_i\}$ for which $\langle d_o, d'_o \rangle \in R_{in}$. Then we have Vi . $\langle d_i, d'_i \rangle \in R$ since R is a simulation, so either $\langle d_i, d'_i \rangle \in R^*_{in} \cup R_{out}$ or for some $j \in J$ $d_i \in C_j$ and $d'_i \in C'_j$ since R respects $\langle \pi_J, \pi'_J \rangle$. Thus (a) follows. cl

Theorem 5.2

(Strong Simulation or $\subseteq$). $D_{in} \times D'_{in}$, $R^*_{out} \subseteq D_{out} \times D'_{out}$ with R^*_{in}, R^{*-1}_{out} both single-valued and total, and $\langle \pi_J, \pi'_J \rangle$ a partition pair of D_{comp}, D'_{comp}, the following two statements are equivalent:

(a) $\hat{\mathcal{A}} = R^*_{in}\, \hat{\mathcal{A}}'\, R^{*-1}_{out}$, and computation sequences $\{d_i\}$ of $\mathcal{A}$, $\{d'_i\}$ of $\mathcal{A}'$ for which $\langle d_o, d'_o \rangle \in R_{in}$ always agree for $\langle \pi_J, \pi'_J \rangle$.

(b) There is a strong simulation $R = R^*_{in} \cup R_{comp} \cup R^*_{out}$ of $\mathcal{A}$ by $\mathcal{A}'$ which respects $\langle \pi_J, \pi'_J \rangle$.

Proof ⇒ (b). By the corresponding proof in Theorem 5.1 there is a simulation $R^*_{in} \cup R_{comp} \cup R_{out}$ which respects $\langle \pi_J, \pi'_J \rangle$ and for which

$$\langle e,e' \rangle \in R_{out} \Rightarrow \text{For some } \langle d,d' \rangle \in R^*_{in},\ e = \hat{a}(d) \text{ and } e' = \hat{a}'(d').$$
$$\Rightarrow \langle e,e' \rangle \in R^*_{out}, \text{ since } \hat{a} = R^*_{in}\ \hat{a}'\ R^{*-1}_{out} \text{ and both}$$
$$R^*_{in} \text{ and } \hat{a}' \text{ are single valued.}$$

Thus $R_{out} \subseteq R^*_{out}$, whence $R^*_{in} \cup R_{comp} \cup R^*_{out}$ is also a simulation respecting $\langle \pi_J, \pi' \rangle$ and moreover a strong one, from the conditions of the Theorem.

(b) ⇒ (a). Take any $d_o \in D_{in}$, and $d'_o = R^*_{in}(d_o)$. This is defined since R^*_{in} is total. Then by Theorem 5.1 the computation sequences $\{d_i\}$, $\{d'_i\}$ agree for $\langle \pi_J, \pi'_J \rangle$. It follows also from this that either both or neither of $\hat{a}(d_o)$, $\hat{a}'(d'_o)$ are defined. If neither, then neither of the functions $\hat{a}$, $R^*_{in}\ \hat{a}'\ R^{*-1}_{out}$ is defined for d_o. If both, then for some k $\langle d_k, d'_k \rangle \in R^*_{out}$ and $d_k = \hat{a}(d_o)$, $d'_k = \hat{a}'(d'_o)$. But $d_k = R^{*-1}_{out}(d'_k)$ and $d'_o = R^*_{in}(d_o)$ so the functions $\hat{a}$, $R^*_{in}\ \hat{a}'\ R^{*-1}_{out}$ both yield result d_k for argument $d_,$ since R^{*-1}_{out} is single valued.

It follows that $\hat{a} = R^*_{in}\ \hat{a}'\ R^{*-1}_{out}$. □

(Note that this proof nowhere uses the totality of R^{*-1}_{out})

Now the coarsest partition pair $\langle \pi_J, \pi'_J \rangle$ has J a singleton, and any simulation respects it. We therefore have the following corollary to' Theorem 5.1.

Corollary 5.3. Given R^*_{in} there is a weak simulation $R = R^*_{in} \cup R_{comp} \cup R_{out}$ of a by a' if and only if computation sequences $\{d_i\}$ in a, $\{d'_i\}$ in a' such that $\langle d_o, d'_o \rangle \in R^*_{in}$ always have lengths either both undefined or equal (the length of $\{d_i\}$ is defined as $\min\{k \mid d_k \in D_{out}\}$). □

There is a corresponding corollary to Theorem 5.2, which we omit.

Finally, we give a corollary for flowchart programs of the same shape.

Corollary 5.4. Let $D_{comp} = N \times E$, $D'_{comp} = N \times E'$; call N the node set (common to $\mathcal{A}$ and $\mathcal{A}'$). Define $\pi_N = \{ (n) \times E \mid n \in N \}$, $\pi'_N = \{ \{n\} \times E' \mid n \in N \}$. Then two computation sequences agree for $\langle \pi_N, \pi'_N \rangle$ exactly when they trace the same node path, and so we have the following:

Given R^*_{in}, there is a weak simulation $R = R^*_{in} \cup R_{comp} \cup R_{out}$ of $\mathcal{A}$ by $\mathcal{A}'$ which respects $\langle \pi_N, \pi'_N \rangle$ if and only if every pair of computation sequences $\{d_i\}$ in $\mathcal{A}$, $\{d'_i\}$ in $\mathcal{A}'$ such that $\langle d_o, d'_o \rangle \in R^*_{in}$ trace the same node path. □

Again, there is a corresponding corollary to Theorem 5.2. If as in Section 4 we exhibit R_{comp} by exhibiting $R_{nn'} \subseteq E \times E'$ for $n, n' \in N$, then "R respects $\langle \pi_N, \pi'_N \rangle$" means $R_{nn'} = \emptyset$, $n \neq n'$. This is the situation in Example 2 of that Section.

6. CONCLUSIONS AND POSSIBLE DEVELOPMENTS

The idea of simulation, which is really an application of the notion of weak homomorphism, is interesting in two ways: theoretically, because it allows one to abstract some irrelevant detail from programs to come closer to a definition of algorithm, and practically because there is a manageable technique for proving simulation between programs, which in some cases may make easier the task of proving a program correct.

There are two possible directions for development. First, we have restricted to a single-valued, total transition function F. The situation looks rather different when we relax these conditions - for example we should consider computation trees rather than sequences. Second, we should consider simulation of parallel programs, and treat programs which perform the same computations but not necessarily in the same sequence as serializations of the same parallel program - or of mutually simulating parallel programs. These extensions may bear the same relation to the work of Manna [5] and Ashcroft and Manna [6] on the correctness of nondeterministic and parallel programs as the present paper bears to Manna's earlier work on serial programs [4].

ACKNOWLEDGEMENTS

This work owes much to Peter Landin who largely pioneered the algebraic approach to programs. This paper is in the spirit of [8], although that paper is concerned with the structure of a single program (as a product algebra) rather than relations between programs. I also had profitable discussions with Peter Landin, Rod Burstall and John Laski.

REFERENCES

[1] Floyd, R.W., "Assigning Meanings to Programs", Proceedings of Symposia in Applied Mathematics, American Mathematical Society, Vol. 19, 19-32 (1967).

[2] Milner, R., "A Formal Notion of Simulation Between Programs", Memo 14, Computers and Logic Research Group, University College of Swansea, U.K. (1970).

[3] Milner, R., "Program Simulation: An Extended Formal Notion" Memo 15, Computers and Logic Research Group, University College of Swansea, U.K. (1971).

[4] Manna, Z., "The Correctness of Programs", J. of Computer and Systems Sciences, Vol. 3, No. 2, 119-127 (1969).

[5] Manna, Z., "The Correctness of Non-deterministic Programs", Stanford Artificial Intelligence Project, Memo AI-95, Stanford University (1969).

[6] Ashcroft, E.A., and Manna, Z., "Formalization of Properties of Parallel Programs", Stanford Artificial Intelligence Project, Memo AI-110, Stanford University, (1970).

[7] Ginzburg, A., Algebraic Theory of Automata, Academic Press (1968).

[8] Landin, P., "A Program-Machine Symmetric Automata Theory", Machine Intelligence 5, ed. D. Michie, Edinburgh University Press, 99-120 (1969).