VERIFICATION CONDITIONS AS REPRESENTATIONS FOR PROGRAMS*

by

M.H. van Emden

Research Report CS-76-03

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

January 1976

VERIFICATION CONDITIONS AS REPRESENTATIONS FOR PROGRAMS

M.H. van Emden*

1. Introduction

An important early contribution to the method of characterizing the semantics of programs by means of the semantics of first-order predicate logic was made by Manna [16]. The following subsequent developments in the theory of programs and in resolution theorem-proving have added a new interest to Manna's results.

1) Improved resolution theorem-provers are able to execute effectively, as a program, the logical sentence representing a program according to Manna's method. The logical sentence is the conjunction of the verification conditions, in the sense of Floyd's proof method, that establish the partial correctness of the program. In practical programming it may well be advantageous to program directly in verification conditions (see [8] for E.W. Dijkstra's work pointing in this direction); hence the interest of running verification conditions directly as a program.

2) Improved models for programs (the "grammar model" [1,5]) give Manna's method a wider applicability.

3) Increased understanding [9] of the semantics of predicate logic has made it possible to relate Manna's results to the more recent fixpoint semantics for programs in general.

This paper is devoted to a discussion of these developments. I will give a definition of grammar-modelled programs in their full generality, although only those, the "flowgraphs", that correspond to the "flowchartable" variety

---

*) Department of Computer Science, University of Waterloo,
Waterloo, Ontario N2L 3G1, Canada

will be used. Floyd's method for proving partial correctness is formulated for flowgraphs and justified by operational semantics. After an interlude leading up to the fixpoint semantics for resolution logic, I will prove that the terminating computations of a flowgraph are characterized by the unsatisfiable instances of a formula containing the conjunction of its verification conditions. This amounts to a denotational semantics for flowgraphs. The final section explains how verification conditions can be run as predicate-logic programs on the PROLOG system. [7,19].

2. Grammar-modelled programs, flowgraphs, and flow diagrams

In order to be able to describe sufficiently precisely a class of programs and their computations, I will use the "grammar model", which is inspired by [1,5].

A _machine_ is defined to be a set of _commands_, each of which is a binary relation over some supposedly given set of _states_. A _program schema_ is a grammar of Chomsky's hierarchy of generative grammars. A _program_ is obtained from a program schema and a machine by identifying each of the nonterminals of the program schema with a command of the machine.

For a given state x_0, a _terminated computation_ of a program is a sequence of pairs $(t_1,x_1),\ldots,(t_n,x_n)$ where $t_1 \ldots t_n$ is a string generated by the program schema from which the program is obtained and where $x_0,\ldots,x_n$ are states such that $(x_{i-1},x_i) \in t_i$ for $i = 1,\ldots,n$. Note that $t_1,\ldots,t_n$ are also commands of the associated machine. The _start state_ of the terminated computation is x_0; its _halt state_ is x_n.

An _interpreter_ for a program is a procedure for constructing a terminated computation for any start state for which such a computation exists.

In this paper I will assume that a machine may be characterized in the following way: a machine contains one of more elementary commands, which may differ from example to example, and a machine is the smallest set closed under ("Peirce") product: if c_1 and c_2 are commands of the machine then their product $c_1;c_2$ is also one; a pair (x,z) of states is in $c_1;c_2$ iff there exists a state y such that $(x,y) \in c_1$ and $(y,z) \in c_2$.

Example 1.1

In the set notation for the commands of this example, the variables u,v,w,u',v',w' are considered to range over the rational numbers.

the set of states = $\{(u,v,w)\} \cup \{(u,v)\} \cup \{(w)\}$

the commands:

set notation for command	"Algol" notation for command
$\{((u,v,w),(u,v',w')): v'=v-1 \ \& \ w'=u \times w\}$	v,w:=v-1,u × w
$\{((u,v,w),(u',v',w)): u'=u \times u \ \& \ v'=v/2\}$	u,v:=u × u,v/2
(E.W. Dijkstra's "parallel assignment")	
$\{((u,v),(u,v,1))\}$	real w:=1
("initializing declaration")	
$\{((u,v,w),(w))\}$	und u,v
("undeclaration")	
$\{((u,0,w),(u,0,w))\}$	v=0
$\{((u,v,w),(u,v,w)): \neg v=0\}$	¬v=0
("guard")	
$\{((u,v,w),(u',v',w)): even(v) \ \& \ u'=u \times u \ \& \ v'=v/2\}$	even(v);u,v:=u × u,v/2
$\{((u,0,w),(w))\}$	v=0;und u,v
("guarded command")	

"Guards" and "guarded commands" were introduced by E.W. Dijkstra [8]. A guard is a command, hence a binary relation over the set of states. In general, a guard is a subset of the identity relation consisting of those pairs where the guard, regarded as a condition, holds. A product g;c, where g is a guard and c is not a guard, is called a guarded command.

the <u>machine</u>: some set closed under product containing the above commands

the <u>program</u> <u>schema</u> =

(nonterminals: {S,P,Q}
,terminals: {W1,VO,UV,VW}
,productions: {S → W1 P
,P → VO
,P → UV P
,P → VW P
}
,start symbol: S
)

the <u>program</u> is obtained by the following identifications:

W1 = (<u>real</u> w:=1)

VO = (v=0; <u>und</u> u,v)

UV = (even(v); u,v:=u × u,v/2)

VW = (v,w:=v-1,u × w)

some "computations":

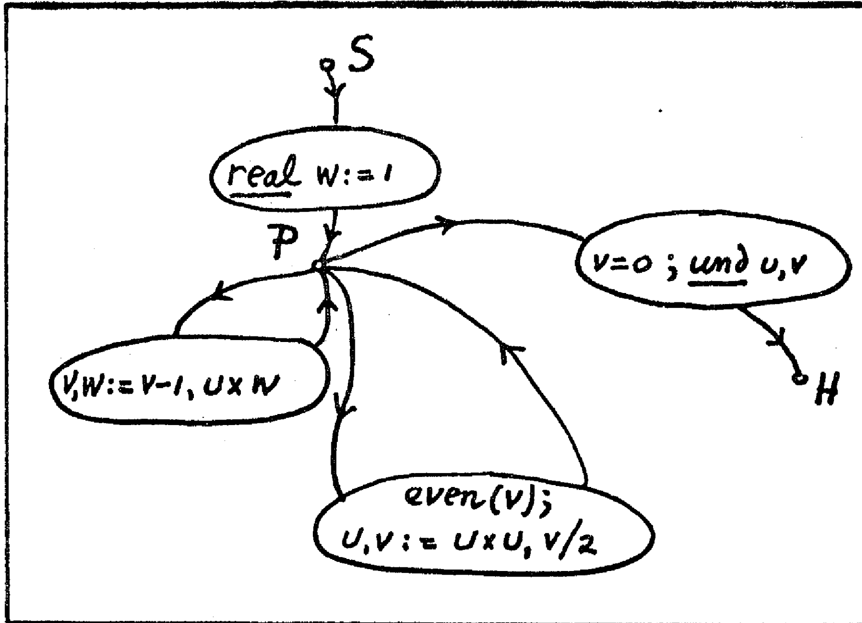
	computation 1		computation 2		computation 3	
i	t_i	x_i	t_i	x_i	t_i	x_i
0	--	(2,10)	--	(2,10)	--	(2,10)
1	W1	(2,10,1)	W1	(2,10,1)	W1	(2,10,1)
2	UV	(4,5,1)	UV	(4,5,1)	UV	(4,5,1)
3	VW	(4,4,4)	VW	(4,4,4)	VW	(4,4,4)
4	UV	(16,2,4)	UV	(16,2,4)	UV	(16,2,4)
5	UV	(256,1,4)	VW	(16,1,64)	VW	(16,1,64)
6	VW	(256,0,1024)	VW	(16,0,1024)	VW	(16,0,1024)
7	VO	(1024)	VO	(1024)	VW	$(16,-1,2^{-14})$
8					VW	$(16,-1,2^{-18})$
						⋮
						ad inf

"Computations" 1 and 2 are terminated computations. They have the same start state and they have the first four pairs in common. "Computation" 3 shares with computation 2 the start state and the first 6 pairs. It is not something defined up till now. This will happen later on, but only for programs derived from schemas of type 3, as the one in the example happens to be.

(end of example 1.1)

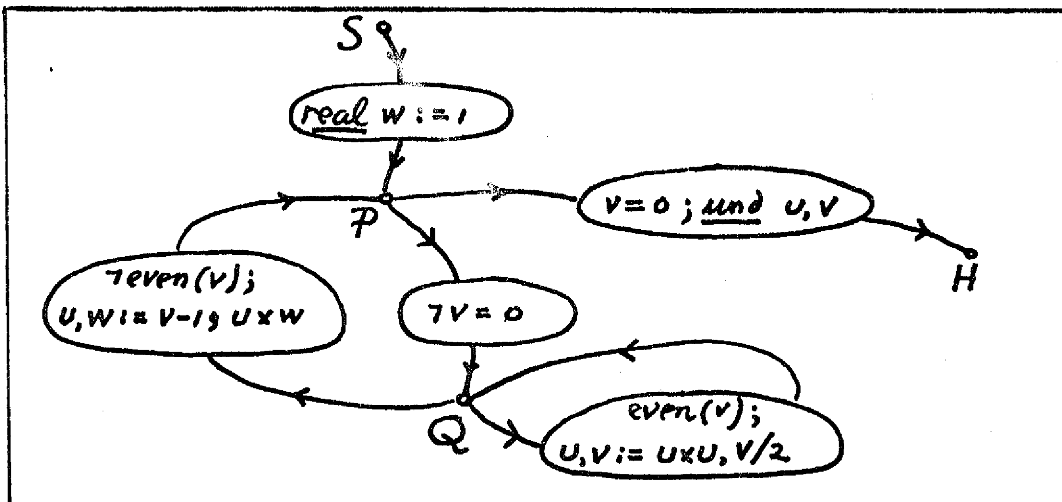
A very convenient representation of a type-3 grammar is the state-transition diagram of the finite automaton which recognizes the language generated by the grammar. The state-transition diagram is a labelled directed graph in which there is a node for each nonterminal in the grammar and one additional node, say H, the halt node. The node corresponding to the start symbol is the start node. For each production $N_1 \rightarrow T\ N_2$ in the grammar there is an arc labelled T from node N_1 to node N_2. For each production $N \rightarrow T$ in the grammar there is an arc labelled T from node N to the halt node H.

The state-transition diagram is a convenient representation because each path through it from the start node to the halt node, if regarded as the sequence of the labels of its arcs, is a string generated by the grammar, and vice versa. A program is conveniently represented by the result of replacing, in the state-transition diagram of a type-3 program schema, the terminal symbols labelling the arcs by commands of an associated machine. This representation, and also the program itself, will be called a flowgraph. For instance, the program in example 1.1 is represented by the labelled graph in Box 1.1



Box 1.1:
The flowgraph
example 1.1

Note that the program in example 1.1 is indeterminate *) : for instance, for the initial state (2,10) there are several different terminated computations, as illustrated in the example. The following program is a determinate variant **) of the one in example 1.1.

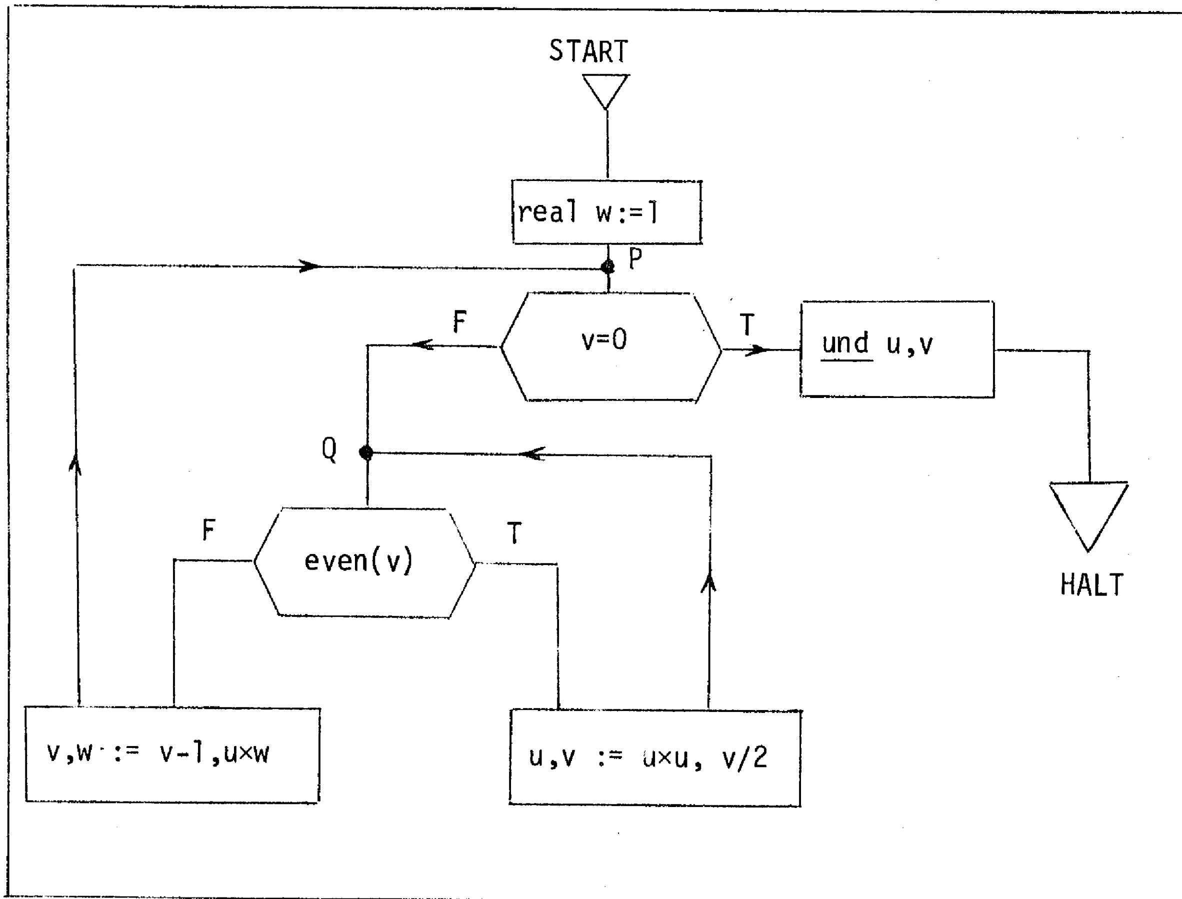


Box 1.2: A determinate variant of Box 1.1

*) The usual "nondeterministic" is an unfortunate (compare "nonfortunistic") and inappropriate (compare "nonappropriistic") neologism, which could only have gained wide acceptance in an illiterate (compare "nonliteristic") discipline.

**) In the sense that the input-output relation computed by the program (see below) is the same.

It is usual in theoretical studies of programming to replace a test by an indeterminate branch and to place complementary partial dummy statements on each outgoing arc of the branch. This device is nearly universally (see for instance [3,6]) attributed to R.M. Karp [11]. If one would eliminate the guards in Box 1.2 by means of Karp's device, the flow diagram in Box 1.3 would be obtained. This relationship is the justification of my choice of the term "flowgraph": it is similar to a flowdiagram, but it is a graph. Flowgraphs can also be found, for instance, in Burstall [6].



Box 1.3: The flowdiagram version of Box 1.2

3. An operational semantics for flowgraphs

A path of a flowgraph is a sequence of (node, state) pairs

$$(N_0,x_0),\ldots,(N_k,x_k),\ldots \qquad k = 0,1,\ldots$$

where $N_0 = S$, the start node, and where each pair after the first is a successor of the pair before it in the sequence: a pair (N_i,x_i) is a successor of (N_{i-1},x_{i-1}) iff $(x_{i-1},x_i) \in t_i$, the command labelling an arc from N_{i-1} to N_i.

The state x_0 is called the start state. In a finite path the last pair may be such that no successor for it exists. If the node in such a last pair is the halt node H, then the path is a terminated path, otherwise it is a blocked path.

Note the following simple connection between a terminated computation for programs in general and a terminated path for programs that are also flowgraphs: by the definition of a terminated path

$$(N_0,x_0),\ldots,(N_n,x_n)$$

there must exist an arc from N_{i-1} to N_i for $i = 1,\ldots,n$. Let t_i be the command labelling this arc. Then

$$(t_1,x_1),\ldots,(t_n,x_n)$$

is the terminated computation for initial state x_0.

The semantics for a flowgraph can be described at several levels of detail. At the most detailed level the behaviour of a flowgraph can be described by the set of paths. At a less detailed level it can be described by the set of (node, state) pairs which occur in some path. Finally, we can regard a flowgraph as a "black box" and only consider "input-output" behaviour. Such behaviour is characterized here by the

input-output relation computed by the flowgraph: a pair (x,y) of states belongs to this binary relation iff there exists a terminated path with x in the first (node,state) pair and y in the last (node,state) pair. Note that the input-output relation, as it consists of pairs of states, has the status of a command. This suggests a hierarchical structure of flowgraphs: a flowgraph can itself be a command in another flowgraph.

Blikle [5] discussed semantics at all three levels of detail; the use of generalized paths ("computation sequences") for the characterization of input-output behaviour of programs with a type-2 schema can be found in de Bakker and de Roever [3].

4. Floyd's method for flowgraphs

Suppose ϕ and ψ are sets of states. Such sets will be, improperly, called assertions, although this term will also be used properly for the statement that a state belongs to such a set. A flowgraph is partially correct with respect to assertions ϕ and ψ if for each terminated path with start state in ϕ, the final state is in ψ. Note that $x \in \phi$ does not imply that there exists a terminated path with x as start state; it is this deficiency that the "partial" in "partial correctness" refers to.

If F is the input-output relation computed by flowgraph $\mathcal{E}$, then its partial correctness with respect to ϕ and ψ can be expressed as an inclusion among binary relations:

$$\phi';F \subseteq F;\psi'$$

where ϕ' is the partial identity containing just those pairs (x,x) such that $x \in \phi$, and similarly for ψ'. In the sequel I suppose the meaning of ";" to be extended so as to include an automatic transition of possibly occurring sets of states to their corresponding partial identities. Thus, the above

property will be written as

$$\phi;F \subseteq F;\psi \qquad \dots (3.1)$$

A convenient notation (when F is not very short) for the partial correctness of $\mathcal{E}$ is the following variant of a notation due to C.A.R. Hoare:

$$\{\phi\}\ \mathcal{E}\ \{\phi\}$$

The method of representing the behaviour of programs and their components as binary relations seems to be due to de Bakker and Scott [4]. The expression (3.1) for partial correctness may be found in [3].

The purpose of the method of Floyd [10] is to prove partial correctness for a program written as a flowdiagram. The method applies to flowgraphs as well, as will now be explained [6]. Let S be the start node and H the halt node of a flowgraph. According to the method there is associated with each node an assertion which is denoted here by the same symbol as the associated node; the context should make it clear which type of object meant. The assertions are said to <u>verify</u> the flowgraph if for each arc the <u>verification condition</u>

$$L_1;C \subseteq C;L_2$$

holds; L_1 (L_2) is the assertion associated with the initial (final) node of the arc, and C is the command labelling the arc.

The premiss of Floyd's rule of proof is that the flowgraph be verified. The conclusion is its partial correctness with respect to any assertions $\phi \subseteq S$ and $\psi \supseteq H$:

$$\phi;F \subseteq F;\psi$$

where F is the input-output relation computed by the flowgraph.

<u>Theorem 4.1</u>. If a (node,state) pair (L,x) occurs in a path with start state $x_0 \in S$ then $x \in L$, the associated assertion.

Proof. Let (L_{i-1},x_{i-1}), (L_i,x_i) be two successive pairs in the path. By the definition of a path, $(x_{i-1},x_i) \in C_i$, the command labelling an arc from L_{i-1} to L_i in the flowgraph. By the supposition that the assertions verify the flowgraph, $L_{i-1};C_i \subseteq C_i;L_i$. Suppose now that $x_{i-1} \in L_{i-1}$, then $x_i \in L_i$. Apparently, if in a path of a verified flowgraph $x \in L$, then the same holds for all subsequent pairs. It was assumed that $x_0 \in S$, the assertion associated with the node in the first pair of a path.

It is easy to see that Floyd's rule of proof is justified by the special case of this theorem for finite paths. It should be noted that Floyd's method may also be usefully applied to algorithms that do not terminate (operating systems, or programs controlling telephone exchanges may be designed never to terminate).

Example 4.1 Let u_0 and v_0 be rational constants and u, v, and w be variables ranging over the rationals. Associate the assertions (sets of states) $\{(u_0,v_0)\}$, $\{(u,v,w):w \times u^v = u_0^{v_0}\}$, $\{(u_0^{v_0})\}$ with the nodes S, P, and H, respectively, of the flowgraph in Example 1.1 or Box 1.1. Then the following verification conditions hold true:

{S} real w := 1 {P}

,{P} v=0;und u,v{H}

,{P} even(v);u,v := u×u,v/2{P}

,and {P}v,w := v-1,u×w{P}

Hence, the flowgraph is verified. Floyd's rule of proof concludes that, whenever an interpreter constructs a terminated computation, the halt state of the computation consists of the singleton $u_0^{v_0}$, where u_0 and v_0 are the values of u and v in the start state of the computation.

5. Resolution Logic

For ease of reference I have collected in this section some definitions and results concerning the clausal form of first-order predicate logic. A useful introduction to clausal form and its use in resolution theorem-proving is [17].

The formation rules of the language are as follows. A sentence is a set of clauses. A clause is a set of literals L_i written as

$$L_1 \ldots L_n$$

except when the set empty: it is then called the null clause and is written as

$$\square$$

A literal is either +A (and then it is a positive literal) or -A (and then it is a negative literal), where A is an atomic formula. An atomic formula is a $P(t_1,\ldots,t_m)$, where P is an m-place predicate symbol and $t_1,\ldots,t_m$ are terms.

A term is either a variable or an expression $f(t_1,\ldots,t_k)$ where f is a k-place function symbol and $t_1,\ldots,t_k$ are terms. Constants are 0-place function symbols. Constants and variables are chosen from the same set of symbols; variables are distinguished by being preceded by an asterisk. The predicate symbols and the function symbols are sets of symbols mutually disjoint from each other and from the constants and from the variables.

One sentence, clause, literal, atomic formula, or term is an instance of another if the one can be the result of substituting a term

for every occurrence of a given variable in the other. A sentence, clause, literal, atomic formula, or term is ground if it contains no variables.

I will explain that part of a semantics for logic in clausal form which assigns relations as denotations to predicate symbols and which determines whether a sentence is true in a given interpretation.

The set of ground terms which can be constructed from the constants and other function symbols occurring in a sentence S is called the Herbrand universe of S. The set of all ground atomic formulas $P(t_1,\ldots,t_m)$, such that P occurs in S and $t_1,\ldots,t_m$ belong to the Herbrand universe H of S, is called the Herbrand base $\hat{H}$ of S. A Herbrand interpretation is a subset of $\hat{H}$. The denotation of an m-place predicate symbol P in a Herbrand interpretation I is the following m-place relation:

$$\{(t_1,\ldots,t_m) : P(t_1,\ldots,t_m) \in I\}$$

The following rules determine whether a sentence S is true in a Herbrand interpretation I. (If this is the case, then I is said to be a (Herbrand) model of S; the set of models of S will be written as **M**(S); if **M**(S) is nonempty then S is said to be consistent).

A ground literal +A is true in I iff $A \in I$.

A ground literal -A is true in I iff $A \notin I$.

A ground clause is true in I iff at least one of its literals is true in I. (Hence a ground clause is to be understood as the disjunction of its literals.)

A clause C is <u>true in I</u> iff every ground instance of C is true in I. (Hence a clause is to be understood as the universally quantified disjunction of its literals.)

A sentence is <u>true in I</u> iff each of its clauses is true in I. (Hence a sentence is to be understood as the conjunction of its clauses.)

It may be proved [9] that for any consistent set of "Horn clauses" (those with at most one positive literal) the intersection of all Herbrand models is itself a model; this one will be called the <u>minimal model</u>. This property of Horn clauses is called the <u>model-intersection property</u>.

For a set R of regular clauses (those with exactly one positive literal) there is a useful characterization of the minimal model as the minimal fixpoint of certain map T, from interpretations to interpretations, associated with R. Let $\hat{H}$ be the Herbrand base of R. T is defined as follows:

T(I) contains a ground atomic formula $A \in \hat{H}$ iff for some ground instance $C\sigma$ of a clause C in R, $C\sigma = +A\ -A_1\ \ldots\ -A_m$ and $A_1,\ldots,A_m \in I$, $m \geq 0$.

It may then be proved [9] that an interpretation is a model of R iff it is closed under T. Hence, the minimal model of R is the least fixpoint of T. It may, moreover, be proved [9] that

$$\cap \mathbf{M}(R) = \bigcup_{n=0}^{\infty} T^n(\phi)$$

where ϕ is the empty subset of $\hat{H}$.

6. Flowgraphs represented in resolution logic

A program schema of type 3 is represented in resolution logic by a set V of regular clauses, each of which represents a production of the program schema. A terminal symbol is represented by a two-place predicate symbol; a nonterminal by a one-place predicate symbol. Throughout this section I will use as predicate symbol the same symbol as for the terminal or nonterminal it represents: the context will make clear which kind of symbol is intended.

A production

$$N_1 \rightarrow T\ N_2$$

of the program schema, where N_1 and N_2 are nonterminals and T is a terminal, is represented in V by the clause

$$-N_1(*x)\quad -T(*x,*y)\quad +N_2(*y).$$

A production

$$N \rightarrow T$$

of the program schema, where N is a nonterminal and T is a terminal, is represented in V by the clause

$$-N(*x)\quad -T(*x,*y)\quad +H(*y).$$

A machine is represented by a set M of clauses of the form

$$+C(x,y)$$

where the constants x and y represent states and where C represents a command of the machine; $+C(x,y) \in M$ iff the pair of states $(x,y) \in C$, a command of the machine.

A program has been defined as the result of identifying each nonterminal in a given program schema with a command in a given machine. The representation in logic is obtained in a similar way: the predicate symbols for nonterminals in V or those for commands in M may be chosen in such a way that the desired identifications take place in the set $V \cup M$, which is then the representation of the resulting program. Again, it should be clear from the context whether a symbol means a command, the nonterminal identified with it, or the predicate symbol representing either.

Note that a clause $-N_1(*x)$ $-T(*x,*y)$ $+N_2(*y)$ representing a production $N_1 \rightarrow T\ N_2$ may be regarded as a verification condition in the sense of Floyd's method: in a given interpretation the clause is true iff

$$N_1'; T' \subseteq T'; N_2'$$

where N_1', T', N_2' are the denotations of N_1, T, and N_2 respectively in the interpretation.

<u>Example 6.1</u>

The clausal form of the verification conditions in Example 4.1 and also the representation of the program schema in Example 1.1 is:

-S(*x) -W1(*x,*y) +P(*y).

-P(*x) -V0(*x,*y) +H(*y).

-P(*x) -UV(*x,*y) +P(*y).

-P(*x) -VW(*x,*y) +P(*y).

end of Example 6.1.

Let T be the function (defined in the previous section) from interpretations to interpretations associated with the regular clause

$$V \cup M \cup \{+S(x_0) : x_0 \in \varphi\}$$

for some set φ of states.

Lemma 6.1

For n = 1,2,..., the ground atomic formula $N(y) \in T^n(\phi)$ iff the (node, state)-pair (N,y) occurs in a path of length at most n of which the start state $x_0 \in \varphi$.

Proof

Proceed by induction on n. $N(y) \in T(\phi)$ implies, by the definition of T (case m = 0), that N is S and $y \in \varphi$ and that (N,y) is the first pair of a path with start state in φ.

For the induction step, assume the lemma for $n \leq k$. $N(y) \in T^{k+1}(\phi)$ implies, by the definition of T, that, for some $0 \leq k_1 \leq k$, $N(y) \notin T^{k_1}(\phi)$ and $N(y) \in T^{k_1+1}(\phi)$. Again, by the definition of T, this implies the existence of an N_1 and an x such that $-N_1(x)$ $-C(x,y)$ $+N(y)$ is a ground instance of some clause in V, that $+C(x,y) \in M$ and $N_1(x) \in T^{k_1}(\phi)$. By the induction assumption (N_1,x) is in a path of length at most k_1; (N,y) is a successor of (N_1,x) and occurs therefore in a path of length at most k_1+1.

This completes the "only if" part of the lemma. The converse may be proved in similar way.

The fact that $\bigcup_{n=0}^{\infty} T^n(\phi)$ is the minimal model $\cap \mathbf{M}(P)$ of $P = V \cup M \cup \{+S(x_0) : x_0 \in \varphi\}$ allows us to derive from lemma 6.1 the following

Theorem 6.1

A ground atomic formula $N(y) \in \cap \mathbf{M}(P)$ iff the (node, state)-pair (N,y) is in some path with start state in φ.

It is now possible to obtain a characterization of termination of program execution by means of unsatisfiability which is quite similar to Manna's [16]:

Corollary 6.1

$P \cup \{-H(xy)\}$ is unsatisfiable iff there is a terminated path with start node in φ.

Proof

According to "Herbrand's theorem" (see, for instance, [18]) $P \cup \{-H(xy)\}$ is unsatisfiable iff a finite set of ground instances of clauses in $P \cup \{-H(xy)\}$ is unsatisfiable. I will prove first that there exists such a set with only one ground instance of $-H(xy)$ in it. Let $-H(y_1),\ldots,-H(y_n)$ be the ground instances in a set S that must exist by Herbrand's theorem. At least one, say $H(y_k)$, of $H(y_1),\ldots,H(y_n)$ must be in the minimal model $\cap \mathbf{M}(P_1)$ of P_1, the subset of S of ground instances of clauses in P. We know that

$$\mathbf{M}(P) \subseteq \mathbf{M}(P_1)$$

and, hence,

$$\cap \mathbf{M}(P_1) \subseteq \cap \mathbf{M}(P).$$

Therefore, $H(y_k) \in \cap \mathbf{M}(P)$ and by theorem 6.1 (H,y_k) is in a path with start state in φ. That path is a terminated path.

This completes the "only if" part of the corollary. The converse may be proved in similar way.

Manna [16] considers a formula ("W_{AP}") which is analogous to

$$V \cup \{+S(x_0)\} \cup \{-H(xy)\}$$

where interpretations of predicate symbols representing terminals are fixed a priori. I achieve the same effect by explicitly listing, as it were, those interpretations in M, and then adding this set of clauses.

Theorem 6.2

The flowgraph represented by $V \cup M$ is partially correct with respect to assertions φ and ψ iff $V \cup M$ has a model with $\varphi \subseteq S'$ and $\psi \supseteq H'$, where S' and H' are the denotations of S and H in this model.

Proof

To show that the flowgraph is partially correct, suppose that

$$(S,x_0),\ldots,(H,x_n)$$

is a terminated path and that $x_0 \in \varphi$. Then, by Theorem 6.1 the ground atomic formulas

$$S(x_0),\ldots,H(x_n)$$

are in the minimal model of P and therefore also in the supposedly existing model of $V \cup M$ with $\varphi \subseteq S'$ and $\psi \supseteq H'$. Therefore, $x_n \in \psi$ which establishes partial correctness.

To prove the "only if" part I show that $\cap \mathbf{M}(P)$ is a model of $V \cup M$ with $\varphi \subseteq S'$ and $\psi \supseteq H'$. Any model of P (and because P contains only Horn clauses $\cap \mathbf{M}(P)$ is one by the model-intersection property) is a model of $V \cup M$ with $\varphi \subseteq S'$.

It remains to show that $H' \subseteq \psi$. By Theorem 6.1, $H(x) \in \cap \mathbf{M}(P)$ only for those x which are the halt state of a terminated path of which the initial state $\in \varphi$. By the assumption of partial correctness, $x \in \psi$. Therefore, $H' \subseteq \psi$.

Theorem 6.2 justifies in its "if" part Floyd's method of proof by the denotational semantics for flowgraphs as given by Theorem 6.1. The "only if" part may be regarded as a completeness result for the method: if a flowgraph is in fact partially correct then assertions exist that would prove it (if found) according to Floyd's rule of proof. For the completeness property see de Bakker and Meertens [2] who established it for a generalization of Floyd's method that applies to programs derived from schemas of type 2.

7. Running verification conditions as predicate-logic programs

The termination result in corollary 6.1 is different from the one obtained by Manna [16], who finds that unsatisfiability of a formula with a free variable representing input is equivalent to termination for all inputs. The representation of a flowgraph as a sentence, as opposed to a formula with a free variable, has each admissible input represented by unique constant, say, x_0, which is explicitly present in a separate clause $+S(x_0)$. The termination result therefore applies to just the inputs thus listed in the representing sentence.

In C_{2i+1} the selected literal is $-T(x_1,xy)$ and therefore only input resolution is possible, to wit, against a clause $+T(x_i,x_{i+1})$ of M, which explains the form of C_{2i}, with which again only input resolution is possible, this time against a clause of V, which explains the form of C_{2i+1}.

It is not an isolated phenomenon that there is no occasion for ancestor resolution in SL-derivations from a sentence characterizing the semantics of a flowgraph. This is also the case for sentences containing only Horn clauses and derivations where the top clause contains negative literals only. Kowalski's "procedural interpretation" [13,14] is a powerful guide for the use of such clauses in a heuristically effective way.

"Predicate-logic programming", as it now exists, can be characterized as the endeavour of exploiting in combination the heuristic power of Horn clauses and the fact that an SL-resolution theorem-prover, without the need for autonomous ancestor resolution, can be efficiently implemented. The PROLOG system for predicate-logic programming, developed by Colmerauer and his group in the University of Aix-Marseille, is basically such a theorem-prover [7,19].

```
-TTY-BOOLISTE-LIREFICHIER-BOOLISTE-TTY.       ]-PROLOG job control
+.(DG,1).

     -S(*X) -W1(*X,*Y) +P(*Y)   -SORTER(*Y) -LIGNE.
     -P(*X) -V0(*X,*Y) +H(*Y)   -SORTER(*Y) -LIGNE.
V  - -P(*X) -UV(*X,*Y) +P(*Y)   -SORTER(*Y) -LIGNE.   -output routines
     -P(*X) -VW(*X,*Y) +P(*Y)   -SORTER(*Y) -LIGNE.    tracing the compu-
     -H(*Y).                                           tation

     +W1(*U.*V,*U.*V.1).
     +V0(*U.0.*W,*W).
     +EVEN(*V) -RESTE(*V,2,*V1) -EQUALS(*V1,0).
     +UV(*U.*V.*W,*U1.*V1.*W) -EVEN(*V)
M' -                          -MULT(*U,*U,*U1)
                              -DIV(*V,2,*V1).
     +VW(*U.*V.*W,*U.*V1.*W1) -MOINS(*V,1,*V1)
                              -MULT(*U,*W,*W1).
     +EQUALS(*X,*X).

     +FIN.
     =+S(2.10).  <------ PROLOG job control

     1024
     256 . 0 . 1024
     256 . 1 . 4         states of the path in reverse order of the
     16 . 2 . 4          flowgraph represented by V ∪ M
     4 . 4 . 4
     4 . 5 . 1
     2 . 10 . 1
     =-STOP. ----------> PROLOG job control
```

An annotated PROLOG session showing the unsatisfiability of

$V \cup M' \cup \{-H(xy),+S(2.10)\}$ where V is the representation in predicate logic of the program schema in example 1.1, and M' is equivalent to the infinite set M of ground clauses representing the machine in example 1.1. Note that V is also the set of verification conditions required for proving partial correctness (according to Floyd's method) of the flowgraph represented by $V \cup M$.

8. References

1. E. Ashcroft, Z. Manna, A. Pnueli: Decidable properties of monadic functional schemas. Journal ACM 20 (1973) 489-499.

2. J. W. de Bakker and L. G. L. Th. Meertens: On the completeness of the inductive assertion method. Report IW12, Mathematical Centre, Amsterdam, 1973.

3. J. W. de Bakker and W. P. de Roever: A calculus for recursive program schemas. In M. Nivat (ed): Automata, Languages, and Programming, North Holland, 1973.

4. J. W. de Bakker and Dana Scott: A theory of programs, an outline of joint work. IBM Seminar, Vienna, August 1969.

5. A. Blikle: An extended approach to the mathematical analysis of programs. CC PAS Report 169 (1974), Computation Centre Polish Academy of Sciences, Warsaw PKiN, P.O.Box 22, Poland.

6. R. M. Burstall: An algebraic description of programs with assertions, verification, and simulation. SIGPLAN notices 7 (1972), pp 7-14.

7. A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel: Un système de communication homme-machine en français. Rapport préliminaire, Groupe de Recherche en Intelligence Artificielle, U.E.R. de Luminy, Marseille, 1972.

8. E. W. Dijkstra: Guarded commands, nondeterminacy and formal derivation of programs. CACM, 18 (1975), 453-457.

9. M. H. van Emden and R. A. Kowalski: The semantics of predicate logic as a programming language. Report MIP-R-103, Dept. of Machine Intelligence, University of Edinburgh, 1974.

10. R. W. Floyd: Assigning meanings to programs. In J. T. Schwartz (ed.): Proc. Symposium App. Math. Vol. XIX, Am. Math. Soc., Providence, 1967.

11. R. M. Karp: Some applications of logical syntax to digital-computer programming. Thesis 1959, Harvard University.

12. R. A. Kowalski: A proof procedure using connection graphs. Journal ACM 22 (1975), 572-595.

13. R. A. Kowalski: Predicate logic as programming language. Proc. IFIP 74, pp 569-574, North Holland, 1974.

14. R. A. Kowalski: Logic for problem-solving. DCL Memo 75, Dept. of Artificial Intelligence, University of Edinburgh, 1974.

15. Robert Kowalski and Donald Kuehner: Linear resolution with selection function. Artificial Intelligence 2 (1971), pp 227-260.

16. Zohar Manna: Properties of programs and the first-order predicate calculus. Journal ACM 16 (1969), pp 244-255.

17. N. J. Nilsson: Problem-solving methods in artificial intelligence. McGraw-Hill, New York, 1971.

18. J. A. Robinson: A review of automatic theorem-proving. In J. T. Schwartz (ed.): Proc. Symp. Appl. Math. Vol. XIX, Am. Math. Soc., Providence, 1967.

19. P. Roussel: PROLOG- manuel d'utilisation. Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Marseille 1975.

9. Acknowledgements

Keith Clark pointed out to me the interconnections between Manna's method, verification conditions, and predicate-logic programming. Without his impetus the research reported here would not have started. Discussions with Hajnal Andréka, Alain Colmerauer, and István Németi have sustained the work. Critical remarks by Robert Kowalski have helped in finishing it.

The UK Science Research Council and a Research Grant from the University of Waterloo have provided support.