Consequence Verification of Flowcharts

K.L. Clark
Queen Mary College and Imperial College
University of London

M.H. van Emden
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

CS-79-23

June 1979

Consequence Verification of Flowcharts

K. L. Clark
M. H. van Emden

1 Introduction

For program verification a multitude of different formalisms and proof rules have been proposed. As examples of proof rules we mention McCarthy's recursion induction [14], Scott's induction rule [2], Park's fixpoint induction [17], and subgoal induction [16]. Floyd's method of inductive assertions [9] is based on a proof rule which concludes partial correctness from the truth of verification conditions. Hoare [10] proposed a separate proof rule for each of a selected set of constructs of an Algol-like language. Verification-oriented languages such as Lucid [1] and Euclid [13] come with their own inference systems.

Prior to all of these, Tarski has established [19] a 'methodology of the deductive sciences' which incorporates predicate logic as the sole application-independent inference system, all application-specific matters being embodied in the axioms of the 'deductive theory'. At first sight, programs and their verifications would appear to be a suitable subject for a deductive theory. Yet, although predicate logic plays a part in some approaches to program verification, such as Floyd's, Hoare's and Park's, even these do not follow Tarski's paradigm because they use application-specific rules of inference.

The realisation that first order logic can be used as a programming notation [11,7], with resolution inference systems as program executors, has made it possible to bring program verification within the compass of Tarski's paradigm [5,6]. A logic program comprises a set of first order sentences. These are used to (computationally) infer instances of the relations they describe. To verify the program, to show that all the derivable instances are instances of the relations we want to compute, we just have to show that the program is made up of true statements about these relations. We do this formally by showing that each statement of the program is a logical consequence of set of axioms that constitute an intuitively correct characterization of the relations in question, by showing that the logic program comprises a set of theorems of a theory of what is to be computed. We call this the consequence verification method for logic programs.

In this paper we show how the verification of flowchart programs can be reduced to consequence verification of logic programs. We do this by reading off, from a flowgraph version of the flowchart, an equivalent logic program; equivalent in the sense that the set of input-output pairs computed by the flowchart is exactly the set of instances, of some particular relation, derivable from the logic program. A consequence verification of the logic program is a (partial) verification of the flowchart.

We shall see that there are two straightforward methods for reformulating a flowgraph program as a logic program. The first method gives us a 'forward' description of the flowgraph, a description in terms of the relations computed between the start node and each node of the flowgraph. The second method is the 'backward' description, a description in terms of the relations computed between each node and the halt node. Each of these is an implicit, or recursive, description. A consequence verification of the forward description program is equivalent to a verification of the flowgraph using Floyd's inductive assertions. A consequence verification of the backward description is equivalent to a verification of the flowgraph by subgoal induction.

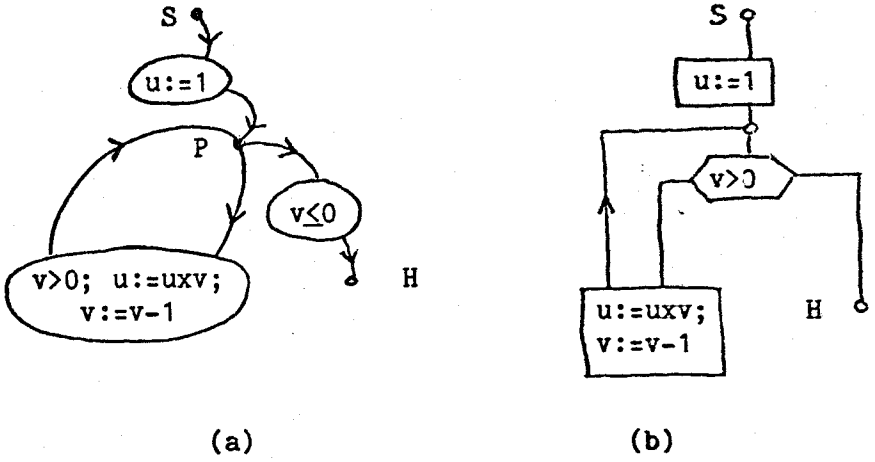
That a flowchartable program can be given two quite different recursive descriptions is not a new idea. In [3] de Bakker gives these alternative representations as two different sets of mutually recursive procedures, and in [4,7] they are given as alternative sets of relational equations. Blikle[4] calls the 'forward' equations the initiation semantics, and the 'backward' equations the continuation semantics. He goes on to show that a verification of the forward equations by fixed point induction[9] corresponds to a Floyd verification of the program, although he does not pursue the verification of the backward equations. We are simply going one step further. A fixed point induction for Blikle's forward equations is a consequence verification of our forward description, which, in turn, reduces to a proof of the verification conditions for Floyd's method. Correspondingly, a fixed point induction for his backward equations is a consequence verification of our backward description, which reduces to a proof of the verification conditions for subgoal induction.

Because of the above correspondences consequence verification does not provide us with a new verification method. Its main interest lies in the fact that it can be justified solely in terms of the model theory of first order logic. Moreover, the manner in which it subsumes Floyd verification and subgoal induction serves to illuminate their essential similarity. We shall see that they differ only in the way in which they implicitly factor the overall computation into component sub-computations.

We offer consequence verification as a conceptual tool, and we offer our analysis of the relationship between it and the other verification methods as a tutorial. We want to demonstrate that the semantics and proof theory of first order logic provide a useful framework for the discussion of computation and verification.

2 Flowgraphs

In this section we define flowgraphs, which are a streamlined version of the conventional flowcharts. An advantage of a flowgraph is that tests and actions are both viewed as binary relations over states; therefore, they are not distinct entities as they are in flowcharts. Each half of a test in a flowchart is represented in a flowgraph, independently of the other half, as a complementary subset of the identity relation. Fig. 1(a) is an example of a flowgraph. Fig. 1(b) is an equivalent flowchart. Each arc of the flowgraph is labelled with a binary relation, called a command, over the states of the computation. In this example the states are pairs of integers. In general, a state is an ordered tuple of the values of the variables accessible to the



(a) (b)

Fig.1

flowgraph. In the flowgraph of Fig. 1

"u:=1" names the binary relation {(u:v, u':v') : u'=1 & v'=v}

"v≤0" names the binary relation {(u:v, u':v') : u'=u & v'=v & v≤0}

The semicolon between binary relations is the usual relational product. A pair (x,y) is in r;r' iff there is a state z such that (x,z) is in r and (z,y) is in r'. Hence,

"v>0; u:=uxv; v:=v-1" denotes {(u:v, u':v') : v>0 & u'=uxv & v'=v-1}.

We may visualize a forward computation of a flowgraph as a path from the start node to the halt node as it is traced by a token carrying a state. The constraint which determines which paths are computations is that an arc can only be traversed if the state is in the domain of the relation labelling the arc. When the arc is traversed, the state changes to one of the values allowed by the relation.

A more precise description of flowgraphs and their computations may be given as follows.

DEFINITIONS
(1) A flowgraph is a labelled, directed graph with one node S (the start node) which has no incoming arc and one node H (the halt node) which has no outgoing arc. Each arc is labelled with a binary relation over a set of states.

(2) A forward computation is a sequence of (node, state) pairs. The set of forward computations of a flowgraph G is the smallest set such that

a) it contains the unit sequence (S,x) for all states x.

b) if it contains (S,x),..., (P,y) and if G has an arc from P to Q labelled with C such that there is some z with (y,z) in C, then it contains also (S,x),...,(P,y),(Q,z).

(3) A backward computation is a sequence of (node,state) pairs. The set of backward computations of a flowgraph G is the smallest set such that

a) it contains the unit sequence (H,x) for all states x.

b) if it contains (Q,y),...,(H,x) and if G has an arc from P to Q labelled with C such that there is some z with (z,y) in C, then it contains also (P,z),(Q,y),...,(H,x).

(4) The input-output relation of the forward computations that end at P is the set of state pairs (x,y) such that

(S,x),....,(P,y)

is a forward computation. Similarly the input-output relation of the backward computations that begin at P is the set of pairs (y,z) such that

(P,y),....,(H,z)

is a backward computation.

(5) The input-output relation of the flowgraph is the input-output relation of the forward computations that end at H, or the input-output relation of the backward computations that begin at S.

The alternative characterisations of the input-output relation of the entire flowgraph given in the last definition are a consequence of the fact that a sequence

(S,x),.....,(H,z)

is a forward computation iff it is a backward computation. Our alternative mappings of flowgraphs into logic programs exploit the fact that we can view the input-output relation of the flowgraph as either the input-output relation of a special subset of the forward computations, or that of a special subset of the backward computations.

3 Logic Program Descriptions of a Flowgraph

Forward Description

Suppose that we associate with each node P of the flowgraph a predicate FP understood as the name of the input-output relation of all the forward computations that end at P; that is, FP(x,y) is true iff there exists a forward computation (S,x),...,(P,y).

Now suppose that there is an arc in the flowgraph from P to Q labelled with a command C. The following implication is a true statement about the relations FP and FQ:

$$(\forall x,y,z)[FQ(x,z) \leftarrow FP(x,y) \;\&\; C(y,z)] \qquad (3.1)$$

We can read the implication as:

for all x,y,z, if y is a state reached at P in a forward computation that starts with x, and z is related to y by command C, then z is a state reached at Q in a forward computation that starts with x.

If S is the start node of the flowgraph, then FS is the input-output relation of all the forward computations that start and end at S. Hence (Vx)FS(x,x) is true.

DEFINITION
The forward description[7] of a flowgraph is a set of implications of the form (3.1), one for each arc of the flowgraph, together with the assertion (Vx)FS(x,x).

Note that the forward description is just a formalisation of the two conditions that characterise the set of forward computations.

Example

The sentences:

FS(x,x),

FP(x,y) <- FS(x,y) & A(y,z),

FP(x,z) <- FP(x,y) & B(y,z),

FH(x,z) <- FP(x,y) & C(y,z),

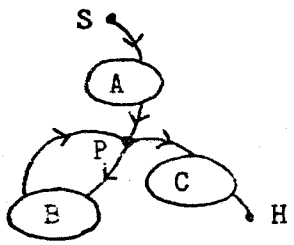


Fig. 2

each implicitly universally quantified with respect to its variables, is the forward description of the flowgraph of Fig. 2. Note that FH, being the name of the input-output relation of the forward computations that end at H, names the input-output relation of the entire flowgraph.

Suppose that C is a set of assertions about the commands of a flowgraph which includes C(a,b) iff (a,b) is in the command relation C. Then the forward description and the set of assertions C characterise the input-output relations of the forward computations in the following sense.

Theorem 1
Let F be the forward description of a flowgraph and let C be the set of command assertions for the flowgraph. Then
F,C FP(x,y)
iff there exists a forward computation (S,x),...,(P,y).

Corollary
F,C FH(x,y)
iff (x,y) is in the input-output relation of the flowgraph.

Sketch Proof

By an induction on the length of the derivation we can show that FP(x,y) is derivable from F and C by a hyper-resolution proof [18] (of length n) iff there is a forward computation (S,x),...,(P,y) (of length n). The hyper-resolution proof simulates the forward computation of the flowgraph. By the soundness and completeness of hyper-resolution FP(x,y) is so derivable iff it is a logical consequence of F and C.

Backward Description

Again we associate with each node P of the flowgraph a predicate, BP, which this time is intended to name the input-output relation of all the backward computations that begin at P. That is, BP(x,y) is true iff there is a backward computation (P,x),...,(H,y).

For each arc from P to Q labelled with command C we read off the implication

$$(\forall x,y,z)[BP(x,z) \leftarrow C(x,y) \,\&\, BQ(y,z)] \qquad 3.2$$

We can read this as the statement:

for all x,y,z, if z is the output of a backward computation that starts at Q with state y, and x is related to y by C, then z is also the output of a backward computation that starts at P with state x.

DEFINITION

The backward description[7] of a flowgraph is a set of implications of the form (3.2), one for each arc of the flowgraph, together with the assertion $(\forall x)BH(x,x)$, where BH is the predicate associated with the start node.

Again the backward description is just the formal statement of the two conditions that characterise the set of backward computations.

Example

The sentences:

$BH(x,x)$,

$BS(x,z) \leftarrow A(x,y) \,\&\, BP(y,z)$,

$BP(x,z) \leftarrow B(x,y) \,\&\, BP(y,z)$,

$BP(x,z) \leftarrow C(x,y) \,\&\, BH(y,z)$.

are the backward description of the flowgraph of Fig. 2. This time BS is the name of the input-output relation of the flowgraph.

Theorem 2

Let B be the backward description of a flowgraph and let C be the set of command assertions of the flowgraph. Then

$$B,C \models BP(s,t)$$

iff (s,t) is in the input-output relation of the backward computations that

begin at P.

Corollary

$$B,C \models BS(s,t)$$

iff (s,t) is in the input-output relation of the flowgraph.

Sketch Proof

Analogous to the proof of Theorem 1 but using SL-resolution [12] as the logic program executor. An SL proof that (u)BP(s,u), using the appropriate selection rule, will simulate the forward construction of the sequence
(P,s),.....,(H,t).

4 Consequence Verification of Logic Programs

The forward description together with the command assertions can be considered a logic program for the predicate FH. By the corollary of Theorem 1 we know that every assertion FH(s,t) which is a logical consequence of the program names an input-output instance of the corresponding flowgraph, and vice versa. Similarly, the backward decription together with the command assertions can be considered a logic program for the predicate BS, and again the set of assertions BS(s,t) derivable from this program name exactly the input-output instances of the flowgraph.

To verify a logic program, to show that each of the assertions R(s,t) that are derivable from the program denote a positive instance of some actual relation R, we need only check that each sentence of the program is a true statement when we interpret the predicate R as the name of the relation R. If the program sentences are 'obviously' true statements about R, this is the end of the matter. If not, we need to prove that they are true of R. This can be done by deriving each sentence of the program from a formal specification of the relation R. This formal specification is a set of sentences of first order logic which make use of the predicate R. However, unlike the sentences of the logic program, they are all obviously true statements when R is interpreted as the name of R.

Fig. 4 is a diagrammatic representation of this verification method[6] for logic programs, which we shall call consequence verification.

Relation to Fixpoint Induction

We can interpret consequence verification as fixpoint induction [17]. We can associate with a logic program comprising a set of simple implications, like those of the forward and backward descriptions, a monotonic mapping from interpretations (a la Tarski semantics) to interpretations. A fixed point of this mapping is a model of the program [8]. Our derivation of the logic program from the specification is a proof that the relations as described by the specification, which are a model of the specification, are also a model of the program. Hence, they comprise a fixed point of the mapping associated with the program. Fixed point induction tells us that the relations computed

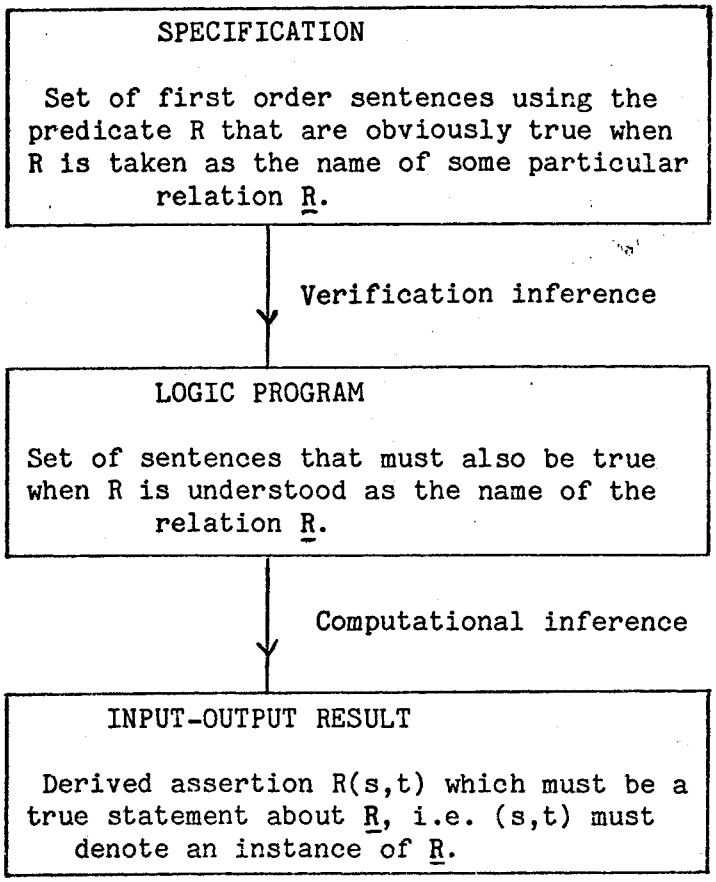


Fig. 4.

by the program are sub-relations of any fixed point interpretation. In other words it confirms that the relation instances derivable from the program are instances of the relations described by the specification.

The consequence verification method for logic programs, and its relation to fixpoint induction, is more fully discussed in [6]. However, as a verification method, it has its own justification in terms of the model theory of first order logic. We express this formally in the following theorem.

Theorem

If the set of sentences of a logic program L are each derivable from a set of specification axioms S, then any relation instance R(s,t) derivable from L denotes an instance of the relation assigned to R in any model of S.

Corollary 1

If the forward description program F, C of a flowgraph is a set of theorems derivable from a specification S, then each input-output instance (s,t) of a flowgraph computation is an instance of the relation assigned to FH in any model of S.

Corollary 2

If the backward description program B,C of a flowgraph is a set of theorems derivable from a specification S, then each input-output instance (s,t) of a computation of the flowgraph is an instance of the relation assigned to BS in any model of S.

Proof

The theorem is just the statement that first order deduction is sound, that it preserves truth. Corollory 1 follows from this fact and Theorem 1, Corollory 2 from this fact and Theorem 2.

The specifications from which we shall try to derive the forward and backward logic programs will implicitly include all the command assertions. So we shall only need to derive the sentences of the forward and backward descriptions of the flowgraph. We shall give explicit definitions of the relations that we believe are computed between the input node and each other node, in the case of the forward description program, and the relations that we believe are computed between the other nodes and the halt node in the case of the backward description program. In addition we shall include any axiom that describes a property of the command relations that we know to be true.

5 Consequence Verification of a Flowgraph: The Forward Case

The flowgraph of Fig. 1(a) has the forward description

FS(u":v",u"v")

FP(u":v",u:v) <- FS(u":v", u':v') & (u =1 & v = v') (5.1)

FP(u":v",u:v) <- FP(u":v",u':v') & (v'>0 & u=u'xv' & v=v'-1)

FH(u":v",u:v) <- FP(u":v",u':v') & (v'≤0 & u=u' & v=v')

Remember that FH is the name of the input-output relation computed by the flowgraph and FP the name of the input-output relation of all forward computations that begin at S and end at P. We would to like prove that at the end of the computation the u component of the memory is the factorial of the v component of the initial state. To do this we take as our specification of FH the definition

FH(u":v", u:v) <-> u = v" !
where (5.2)
0!=1
v!=vx(v-1)! <- v>0

Notice that we have specified the input-output relation in terms of the primitives of the commands of the flowgraph.

To this definition we must add definitions for FS and FP. The input-output relation of the forward computations that end at S is the identity relation, and all the input-output instances of the forward computations that end at P are instances of the relation:

$$\{(u":v",u:v) : uxv!=v"!\} .$$

We therefore add the following definitions to our specification:

$$FS(u":v",u:v) \leftrightarrow u=u" \ \& \ v=v" \qquad (5.3)$$

$$FP(u":v",u:v) \leftrightarrow uxv!=v"!$$

Remember we can augment the specification with any axioms that are true of the command primitives of the flowchart. For the deriviation of (5.1) from specification axioms (5.2) and (5.3) we can make use of algebraic laws such as

$$\begin{aligned} 1xu &= u \\ (uxv)xw &= ux(vxw) \end{aligned} \qquad (5.4)$$

which we know to be true of the program primitives.

Example derivation

Each of the sentences of the forward description (5.1) is a theorem of the specification comprising the definitions and axioms of (5.2),(5.3) and (5.4). As an example we shall show that the third sentence of the forward description is a theorem. Its proof is a demonstration that the relation FP as characterised by the definition of (5.3) satisfies this 'implication equation' of the forward description.

The premiss of the implication is:

$$FP(u":v",u':v') \ \& \ v'>0 \ \& \ u=u'xv' \ \& \ v=v'-1$$

$\leftrightarrow$ $u'xv'!=v"!$ & $v'>0$ & $u=u'xv'$ & $v=v'-1$ (by definition of FP)

$\leftrightarrow$ $u'x(v'x(v'-1)!)=v"!$ & $v'>0$ & $u=u'xv'$ & $v=v'-1$ (using $v'!=v'x(v-1)!\leftarrow v'>0$)

$\leftrightarrow$ $(u'xv')x(v'-1)!=v"!$ & $v'>0$ & $u=u'xv'$ & $v=v'-1$ (associativity law)

$\leftrightarrow$ $uxv!=v"!$ & $v'>0$ & $u=u'xv'$ & $v=v'-1$ (equality substitution)

$\rightarrow$ $FP(u":v", u:v)$ (by definition of FP)

which is the consequent of the implication.

Relation to Floyd's verification method

The consequence verification of the forward description is essentially a verification of the flowgraph using Floyd assertions[9]. If we take u" and v" as parametric input values of the program variables u and v, then each of the definiens of the definitions for FS, FP and FH are the assertions about the 'states of the computation' that would be attached to nodes S, P and H respectively. Such an annotated flowgraph is depicted in Fig. 5. Moreover, the derivation of each of the sentences of the forward description is just the proof that for each forward path between assertion points the assertion at the beginning of the path 'implies' the assertion at the end of the path taking into account the state transformation of the path. In other words, it is a proof of the Floyd verification condition for each such path.

More generally, the task of showing that some flowgraph is partially correct with respect to input condition $\phi(x)$ and input-ouput relation $\psi(x,y)$, is for us the task of augmenting the specification

$$\begin{aligned} &FS(x,y) \leftrightarrow x=y \\ &FH(x,y) \leftrightarrow [\phi(x) \rightarrow \psi(x,y)] \end{aligned} \qquad (5.5)$$

with suitable definitions for predicates FP1,..,FPn associated with the interior nodes of the flowgraph. The constraint on these definitions is that, together with axioms about the program primitives, they enable us to derive all the implications of the forward description.

Second order formalisation

Let $F_{\phi,\psi}$ be the forward description with all mentions of the predicates FS and FH replaced by appropriate instances of the (5.5) definitions. A consequence verification of $F_{\phi,\psi}$ is, in effect, a proof that

$$(\exists FP1,..,\ FPn)F_{\phi,\psi}$$

is true for some fixed interpretation of the command predicates, ϕ and ψ. This is the second order formalisation of the Floyd verification method which Manna gave in [15].

6 Consequence Verification of a Flowgraph: The Backward Case

The backward description of the flowgraph of Fig. 1(a). is

$$BH(u{:}v,u{:}v)$$

$$BS(u{:}v,u"{:}v") \leftarrow (u'=1 \ \&\ v'=v) \ \&\ BP(u'{:}v',u"{:}v") \qquad (6.1)$$

$$BP(u{:}v,u"{:}v") \leftarrow (v>0 \ \&\ u'=uxv \ \&\ v'=v-1) \ \&\ BP(u'{:}v',u"{:}v")$$

$$BP(u{:}v,u"{:}v") \leftarrow v\leq 0 \ \&\ BH(u{:}v,u"{:}v")$$

This time BS is supposed to name the input-output relation of the entire

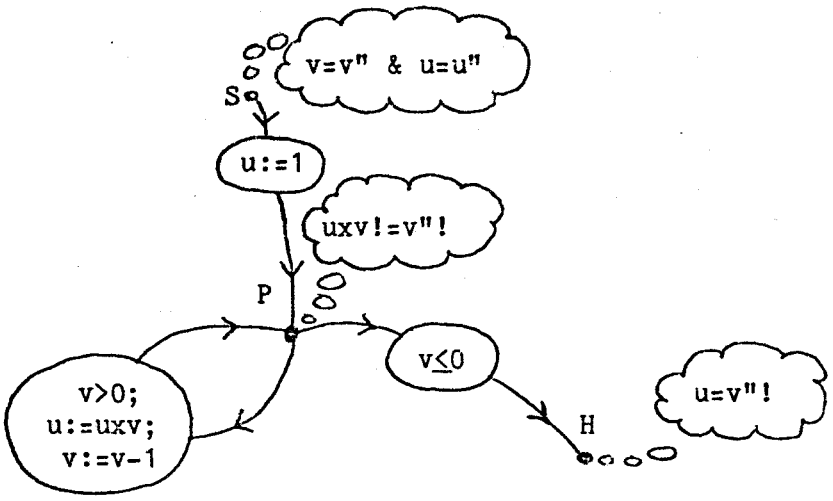


Fig. 5

flowgraph, so its definition is

```
    BS(u:v,u":v") <-> u"=v!
where                                          (6.2)
    0!=1
    v!=vx(v-1)! <- v>0
```

BH names the input-output relations of the backward computations that begin and end at H. We define it as the identity relation. BP names the input-output relation of the flowgraph of Fig. 1(a) minus the initialisation of u to 1. Without this initialisation the flowgraph computes the relation

```
{(u:v,u":v") : u"=uxv!}.
```

Our definitions for BH and BP are:

```
BH(u:v,u":v") <-> u=u" & v=v"
                                               (6.3)
BP(u:v,u":v") <-> u"=uxv!
```

Example derivation

With the addition of certain algebraic axioms about the program primitives each of the sentences of the backward description (6.1) is a theorem of the specification comprising the definitions (6.2) and (6.3). For comparison with the forward description derivation we show that the third sentence of the backward description is a theorem.

The consequent of the implication is:

```
BP(u:v,u":v")

<->u"=uxv!  (definition of BP)

<- v>0 & u"=ux(vx(v-1)!)  (using v!=vx(v-1)! <- v>0)
```

<->v>0 & u"=(uxv)x(v-1)! (associativity law)

<- v>0 & u'=uxv & v'=v-1 & u"=u'xv'! (equality introduction)

<->v>0 & u'=uxv & v'=v-1 & BP(u:v,u':v') (definition of BP)

which is the antecedent of the implication.

Notice that the derivation comprises almost exactly the steps that were needed to derive the corresponding implication of the forward description. However, the former was a forward deduction, inferring the consequent from the antecedent. This is a backward deduction, reducing the consequent to the antecedent. The moral seems to be that the same deductive work is required no matter which relational view we take of the flowgraph. The only significant difference lies in the definitions that we have to invent for the predicates associated with the interior nodes.

Relation to subgoal induction

The three implications of the backward description, specialised by replacing the reference to FS by certain equality substitutions, are exactly the verification conditions that would be generated by the subgoal induction verification method[16]. More interestingly, just as the definiens of the forward description definitions were Floyd assertions describing the states of the computation in terms of parametric initial values u",v" of the program variables, so the definiens of the backward description definitions describe the states of the computation in terms of parametric final values u",v" of the program variables. Fig. 6. is the flowgraph annotated with these assertions. The derivation of each of the sentences of the backward description is now a proof that the assertion at the end of each path between assertions 'implies' the assertion at the beginning of the path taking in to account the inverse of the state transformation for the path. In other words it is a sort of backwards Floyd verification.

More generally, a subgoal verification of the flowgraph with respect to input predicate $\phi(x)$ and input-output predicate $\psi(x,y)$ is for us the task of augmenting the pair of definitions

$$\begin{aligned} &BH(x,y) \leftrightarrow x=y \\ &BS(x,y) \leftrightarrow [\phi(x) \rightarrow \psi(x,y)] \end{aligned} \qquad (6.4)$$

with definitions for predicates BP1,..,BPn associated with the interior nodes. The constraint on these definitions is that, with suitable extra axioms about the command predicates and the correctness predicates and , we are able to derive each of the implications of the backward description.

Second order formalisation

Let $B_{\phi,\psi}$ be the backward description with occurences of BH and BS replaced in accordance with definitions (6.4). The finding of suitable definitions for BP1,..,BPn is a proof that

$$(\exists BP1,..,\ BPn)B_{\phi,\psi}$$

is true for some fixed interpretation of the command predicates, ϕ and ψ. This formula provides us with an alternative second order representation of the concept of partial correctness. It also gives us a second order

formalisation of subgoal induction.

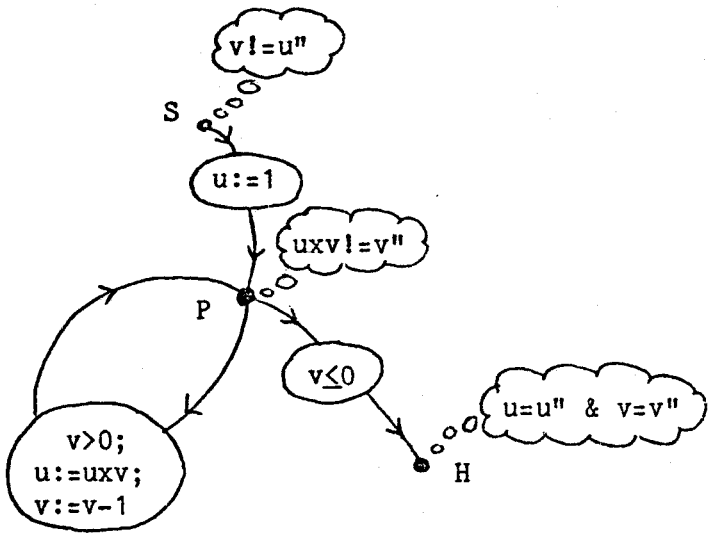


Fig. 6

7 Concluding remarks

We believe that the foregoing analysis does show that the concepts of formal logic provide a rich framework in which to discuss computation and verification. The undertaking of the analysis certainly sharpened the authors understanding of the relationship between Floyd's inductive assertions, subgoal induction, and fixed point induction.

We have seen that, coupled with an appropriate resolution theorem prover, verification conditions can be 'run' as programs. Extending this idea, we can think of the flowgraph as the 'compiled' form of these logic programs. Since logic programs are correct if they comprise a set of (provable) true statements. This suggests what we believe will prove a most fruitful metaphor. It is that programs essentally comprise a set of computationally useful theorems about the relations they compute.

8 Acknowledgements

We gratefully acknowledge the support of the Canadian National Science and Engineering Council and the British Science Research Council. We should also like to thank Cameron Burton for some useful comments on an earlier draft.

9 References

1. E.A. Ashcroft and W.W. Wadge: Lucid, a non-procedural language with iteration. CACM 20 (1977), 519-526.

2. J.W. de Bakker and D. Scott: A theory of programs. IBM Seminar, Vienna 1969.

3. J.W. de Bakker: The fixed point approach in semantics, theory and applications. In Foundations of Computer Science (ed. de Bakker), Tract 63, Mathematics Centrum, Amsterdam, 1975.

4. A. Blikle: A comparative review of some program verification methods. In Mathematical Foundations of Computer Science 1977, (ed. J. Gruska), Springer-Verlag,1977.

5. K.L. Clark and S.Å. Tärnlund: A first-order theory of data and programs. Proc. IFIP 1977, 939-944.

6. K.L. Clark: Predicate logic as a computational formalism, Research Monograph (in preparation), Dept. of Computer Science & Statistics, Queen Mary College, London.

7. M.H van Emden: Relational equations, grammars and programs. Proc. Conf. Theoretical Computer Science, University of Waterloo, 1977.

8. M. H. van Emden and R.A. Kowalski: The semantics of Predicate Logic as Programming Language. JACM 23(4). 1976 pp. 733-742.

9. R.W. Floyd: Assigning meanings to programs. Proc. Symp. App. Math. Vol. XIX (ed. J.T. Schwartz), A.M.S., Providence, 1967.

10. C.A.R. Hoare: An axiomatic basis for computer programming. Comm. ACM 12 (1969), 576-581.

11. R.A. Kowalski: Predicate logic as a programming language. Proc. IFIP 1974, 556-574.

12. R.A. Kowalski and D. Kuehner: Linear resolution with selection function. Artificial Intelligence. 2. 1971, 227-260.

13. B.W. Lampson et al.: Report on the programming language Euclid. SIGPLAN Notices 12 (1977).

14. J. McCarthy: A basis for a mathematical theory of computation. Computer Programming and Formal Systems (eds. P. Braffort and D. Hirschberg), North Holland, 1963.

15. Z. Manna: Second-order mathematical theory of computation, Proc. 2nd Annual ACM Symp. on Theory of Computation, 158-168, 1970

16. J.H. Morris and B. Wegbreit: Subgoal induction. C.ACM 20 (1977), 209-222.

17. D. Park: Fixpoint induction and proofs of program properties. Machine Intelligence 5 (eds. B. Meltzer and D. Michie), Edinburgh University Press 1969.

18. J.A. Robinson: Automatic deduction with Hyper-Resolution Int. J. of Comp. Math. 1. 1965, 227-234.

19. Alfred Tarski: Introduction to Logic and to the Methodology of the Deductive Sciences. Oxford University Press, 1965.

Revised August 13, 1980

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF WATERLOO
TECHNICAL REPORTS 1979

Report No.	Author	Title
CS-79-01*	E.A. Ashcroft W.W. Wadge	Generality Considered Harmful - A Critique of Descriptive Semantics
CS-79-02*	T.S.E. Maibaum	Abstract Data Types and a Semantics for the ANSI/SPARC Architecture
CS-79-03*	D.R. McIntyre	A Maximum Column Partition for Sparse Positive Definite Linear Systems Ordered by the Minimum Degree Ordering Algorithm
CS-79-04*	K. Culik II A. Salomaa	Test Sets and Checking Words for Homomorphism Equivalence
CS-79-05*	T.S.E. Maibaum	The Semantics of Sharing in Parallel Processing
CS-79-06*	C.J. Colbourn K.S. Booth	Linear Time Automorphism Algorithms for Trees, Interval Graphs, and Planar Graphs
CS-79-07*	K. Culik, II N.D. Diamond	A Homomorphic Characterization of Time and Space Complexity Classes of Languages
CS-79-08*	M.R. Levy T.S.E. Maibaum	Continuous Data Types
CS-79-09	K.O. Geddes	Non-Truncated Power Series Solution of Linear ODE's in ALTRAN
CS-79-10*	D.J. Taylor J.P. Black D.E. Morgan	Robust Implementations of Compound Data Structures
CS-79-11*	G.H. Gonnet	Open Addressing Hashing with Unequal-Probability Keys
CS-79-12	M.O. Afolabi	The Design and Implementation of a Package for Symbolic Series Solution of Ordinary Differential Equations
CS-79-13*	W.M. Chan J.A. George	A Linear Time Implementation of the Reverse Cuthill-McKee Algorithm
CS-79-14	D.E. Morgan	Analysis of Closed Queueing Networks with Periodic Servers
CS-79-15*	M.H. van Emden G.J. de Lucena	Predicate Logic as a Language for Parallel Programming
CS-79-16*	J. Karhumäki I. Simon	A Note on Elementary Homorphisms and the Regularity of Equality Sets
CS-79-17*	K. Culik II J. Karhumäki	On the Equality Sets for Homomorphisms on Free Monoids with two Generators
CS-79-18	F.E. Fich	Languages of R-Trivial and Related Monoids

* Out of print - contact author

CS-79-19*	D.R. Cheriton	Multi-Process Structuring and the Thoth Operating System
CS-79-20*	E.A. Ashcroft W.W. Wadge	A Logical Programming Language
CS-79-21*	E.A. Ashcroft W.W. Wadge	Structured LUCID
CS-79-22	G.B. Bonkowski W.M. Gentleman M.A. Malcolm	Porting the Zed Compiler
CS-79-23*	K.L. Clark M.H. van Emden	Consequence Verification of Flow-charts
CS-79-24*	D. Dobkin J.I. Munro	Optimal Time Minimal Space Selection Algorithms
CS-79-25*	P.R.F. Cunha C.J. Lucena T.S.E. Maibaum	On the Design and Specification of Message Oriented Programs
CS-79-26*	T.S.E. Maibaum	Non-Termination, Implicit Definitions and Abstract Data Types
CS-79-27*	D. Dobkin J.I. Munro	Determining the Mode
CS-79-28	T.A. Cargill	A View of Source Text for Diversely Configurable Software
CS-79-29	R.J. Ramirez F.W. Tompa J.I. Munro	Optimum Reorganization Points for Arbitrary Database Costs
CS-79-30*	A. Pereda R.L. Carvalho C.J. Lucena T.S.E. Maibaum	Data Specification Methods
CS-79-31*	J.I. Munro H. Suwanda	Implicit Data Structures for Fast Search and Update
CS-79-32*	D. Rotem J. Urrutia	Circular Permutation Graphs
CS-79-33*	M.S. Brader	PHOTON/532/Set - A Text Formatter
CS-79-34	D.J. Taylor D.E. Morgan J.P. Black	Redundancy in Data Structures: Improving Software Fault Tolerance
CS-79-35	D.J. Taylor D.E. Morgan J.P. Black	Redundancy in Data Structures: Some Theoretical Results
CS-79-36	J.C. Beatty	On the Relationship between the LL(1) and LR(1) Grammars
CS-79-37	E.A. Ashcroft W.W. Wadge	R_x for Semantics

* Out of print - contact author

CS-79-38	E.A. Ashcroft W.W. Wadge	Some Common Misconceptions about LUCID
CS-79-39*	J. Albert K. Culik II	Test Sets for Homomorphism Equivalence on Context Free Languages
CS-79-40	F.W. Tompa R.J. Ramirez	Selection of Efficient Storage Structures
CS-79-41*	P.T. Cox T. Pietrzykowski	Deduction Plans: A Basis for Intelligent Backtracking
CS-79-42	R.C. Read D. Rotem J. Urrutia	Orientations of Circle Graphs

* Out of print - contact author

Revised August 13, 1980

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF WATERLOO

RESEARCH REPORTS 1980

Report No.	Author	Title
CS-80-01	P.T. Cox T. Pietrzykowski	On Reverse Skolemization
CS-80-02	K. Culik II	Homomorphisms: Decidability, Equality and Test Sets
CS-80-03	J. Brzozowski	Open Problems About Regular Languages
CS-80-04	H. Suwanda	Implicit Data Structures for the Dictionary Problem
CS-80-05	M.H. van Emden	Chess-Endgame Advice: A Case Study in Computer Utilization of Knowledge
CS-80-06	Y. Kobuchi K. Culik II	Simulation Relation of Dynamical Systems
CS-80-07	G.H. Gonnet J.I. Munro H. Suwanda	Exegesis of Self-Organizing Linear Search
CS-80-08	J.P. Black D.J. Taylor D.E. Morgan	An Introduction to Robust Data Structures
CS-80-09*	J.Ll. Morris	The Extrapolation of First Order Methods for Parabolic Partial Differential Equations II
CS-80-10+	N. Santoro H. Suwanda	Entropy of the Self-Organizing Linear Lists
CS-80-11	T.S.E. Maibaum C.S. dos Santos A.L. Furtado	A Uniform Logical Treatment of Queries and Updates
CS-80-12	K.R. Apt M.H. van Emden	Contributions to the Theory of Logic Programming
CS-80-13*	J.A. George M.T. Heath	Solution of Sparse Linear Least Squares Problems Using Givens Rotations
CS-80-14	T.S.E. Maibaum	Data Base Instances, Abstract Data Types and Data Base Specification
CS-80-15	J.P. Black D.J. Taylor D.E. Morgan	A Robust B-Tree Implementation
CS-80-16	K.O. Geddes	Block Structure in the Chebyshev-Padé Table
CS-80-17	P. Calamai A.R. Conn	A Stable Algorithm for Solving the Multi-facility Location Problem Involving Euclidean Distances

* Out of print, contact author
+ In preparation

CS-80-18	R.J. Ramirez	Efficient Algorithms for Selecting Efficient Data Storage Structures
CS-80-19	D. Therien	Classification of Regular Languages by Congruences
CS-80-20	J. Buccino	A Reliable Typesetting System for Waterloo
CS-80-21	N. Santoro	Efficient Abstract Implementations for Relational Data Structures
CS-80-22	R.L. de Carvalho T.S.E. Maibaum T.H.C. Pequeno A.A. Pereda P.A.S. Veloso	A Model Theoretic Approach to the Theory of Abstract Data Types and Data Structures
CS-80-23	G.H. Gonnet	A Handbook on Algorithms and Data Structures
CS-80-24	J.P. Black D.J. Taylor D.E. Morgan	A Case Study in Fault Tolerant Software
CS-80-25	N. Santoro	Four O(n**2) Multiplication Methods for Sparse and Dense Boolean Matrices
CS-80-26	J.A. Brzozowski	Development in the Theory of Regular Languages
CS-80-27	J. Bradford T. Pietrzykowski	The Eta Interface
CS-80-28	P. Cunha T.S.E. Maibaum	Resource = Abstract Data Type Data + Synchronization ...
CS-80-29	K. Culik II Arto Salomaa	On Infinite Words Obtained by Interating Morphisms
CS-80-30	T.F. Coleman A.R. Conn	Nonlinear Programming via an Exact Penalty Function: Asymptotic Analysis
CS-80-31	T.F. Coleman A.R. Conn	Nonlinear Programming via an Exact Penalty Function: Global Analysis
CS-80-32	P.R.F. Cunha C.J. Lucena T.S.E. Maibaum	Message Oriented Programming - A Resource Based Methodology
CS-80-33	Karel Culik II Tero Harju	Dominoes Over A Free Monoid
CS-80-34+	K.S. Booth	Dominating Sets in Chordal Graphs
CS-80-35	Alan George J. W-H Liu	Finding Diagonal Block Envelopes of Triangular Factors of Partitioned Matrices
CS-80-36	D.J. Taylor	Robust Storage Structures for Data Structures

+ In preparation
* Out of print, contact author

Research Reports 1980

CS-80-37	R.B. Simpson	A Two Dimensional Mesh Verification Algorithm
CS-80-38†	D.Rotem J. Urrutia	Finding Maximum Qliques in Circle Graphs
CS-80-39†	S.T. Vuong D.D. Cowan	Automated Validation of a Protocol: The CCITT Recommendation X.75 packet level
CS-80-40	F. Mavaddat	Another Experiment with Teaching of Programming Languages
CS-80-41†	K. Culik II J. Pachl	Equivalence problems for mapping on infinite strings
CS-80-42†	J.A. George E. Ng	A comparison of some methods for solving sparse linear least squares problems
CS-80-43†	T.S.E. Maibaim P.R.F. Cunha	Synchronization calculus for message oriented programming

† In Preparation