Department of Applied Analysis
and Computer Science

University of Waterloo

Technical Report CS-73-17

May, 1973

Generalising the Assertion-Method for Proofs of Program Properties

by

E. A. Ashcroft

Generalising the Assertion-Method for Proofs of Program Properties

E. A. Ashcroft
Computer Science
University of Waterloo
Waterloo, Ontario

Abstract

We consider generalising Floyd's assertion-method to more complicated languages than simple flowcharts. We show that the way one associates or attaches assertions to a program, and the intended meaning of these assertions depend on the way the semantics of the program are specified. The criteria for the correctness of the assertion-method, adapted to a particular programming language, depend on the method of semantic definition of the language. We give two general methods of language definition, and show how to obtain sound and adequate adaptations of the assertion-method for any languages defined using these methods.

The paper includes a discussion of language definition methods, and proposes a new method which we call the 'situational' method.

0: INTRODUCTION

The basic idea of Floyd's 'assertion-method' for proving properties of programs [5] is easily understood. Suppose that for some program, which we can think of as a flowchart, we wish to prove that, for all initial values of variables satisfying some'assertion', the final values of the variables (when the program halts) satisfy some other 'assertion'. These 'initial' and 'final' assertions express some desired property of the program, perhaps a mathematical relationship between values, or just a general property like 'the value of x is positive'. We imagine 'attaching' these assertions to the initial and final edges of the flowchart. Now if the program does have the desired property, these attached assertions will be particular examples of 'valid' assertions, that is, assertions which are true of the variable values whenever control passes along the edge to which the assertion is attached (for computations from initial values satisfying the initial assertion). The assertion-method provides a way of proving that a set of assertions, one per edge, are all valid. Thus to prove that our final assertion is valid we must invent other assertions to attach to the other edges in the flowchart, such that the complete set of assertions can be proved valid. These 'intermediate' assertions express properties of the intermediate results of the program, just as the final assertion expresses properties of the final results. To choose these assertions clearly requires intimate knowledge of how the program is supposed to be computing its desired result. The person who will find it easiest to use the method is therefore the person who wrote the program, and stating the intermediate assertions can be incorporated into the documentation of the program, much like 'comments'.

Even considered just as an aid to documentation, the method should have great potential. Unfortunately, there are practical problems in its application. Two of these are 'what sorts of properties can I express naturally as assertions; how formal do I have to be?' and 'how do I apply the method to real programming languages, not just abstract simple flowcharts?' The latter problem is considered in this paper.

The problem has two aspects. Firstly, if there is no flowchart, what are we to attach assertions to, and what does it mean for them to be valid? For example, what would we do with a Lisp program? Secondly, different languages must require different ways of proving the assertions valid. The usual way to prove them valid is to show that they satisfy some condition, a 'verification-condition', derived from the program itself. How do we derive these conditions for different languages?

We show that the answers to both aspects of the problem depend on the <u>method</u> of defining the semantics of the programming language in question. In the first section of the paper we define a very simple programming language in two ways, and show how this results in different versions of the assertion-method. The second section looks in more detail at the methods of defining programming languages, emphasising two methods for which there are easy generalisations of the assertion-method. The third section answers the problem above. We define 'validity' of assertions when using these two methods of language definition, and give the conditions for sound and adequate generalisations of the assertion-method in the two cases. We then show how to actually obtain such methods, by deriving verification-conditions directly from the language definitions.

By bringing out the 'semantics' underlying the various versions of the assertion-method, we obtain a much more unified view. Moreover, the 'meta-theorems' of section three should allow the assertion-method to be easily tailored to specific languages.

In the rest of this introductory section we give a short explanation of the notation for relations used in later sections.

Notation for Relations.

Relations are sets. For $R \subseteq A$, $R(a) \iff a \in R$. Binary relations are written infix : for $R \subseteq A \times B$, $aRb \iff \langle a, b\rangle \in R$.

For set A , E_A denotes the identity over A , i.e. $\{\langle a, a\rangle \mid a \in A\}$.

For $R_1 \subseteq A \times B$, $R_2 \subseteq B \times C$, $R_1 \circ R_2 \subseteq A \times C$ is the usual composition of relations: $aR_1 \circ R_2 c \iff \exists b \in B$, $aR_1 b$ & $bR_2 c$.

$R \circ R$ is written $R^{\textcircled{2}}$, and in general $R^{\textcircled{n}}$ denotes $R \circ R \circ R \circ \cdots \circ R$ (n times). This is in contrast to R^n which denotes $R \times R \times R \times \cdots \times R$ (n times). We distinguish between R^1 and R . R^1 is a set of 1-tuples, which we can think of as <u>singleton sets.</u> Letting 2^R denote the set of all subsets of R , we see that $R^1 \subseteq 2^R$, whereas $R \in 2^R$.

For $R \subseteq A \times B$, and $A' \subseteq A$, $B' \subseteq B$ we define

$$[A']R = \{b \in B \mid aRb \text{ for some } a \in A'\} \subseteq B$$
$$R[B'] = \{a \in A \mid aRb \text{ for some } b \in B'\} \subseteq A .$$

For singleton sets we usually write $[a]R$ and $R[b]$ instead of $[\{a\}]R$ and $R[\{b\}]$. (This causes no confusion even when $R \subseteq 2^A \times B$; For $X \subseteq A$, $[X]R$, denotes $[\{X\}]R$, i.e. $\{b \in B \mid XRb\}$. Note that $\{b \in B \mid \{x\}Rb \text{ for some } x \in X\}$ is $[X^1]R$, not $[X]R$.)

We can use the same notation applied to components of the left and right 'arguments' of R . For $R \subseteq (A \times B) \times (C \times D)$, where $A \cap B = C \cap D = \phi$ (the empty set), if $A' \subseteq A$ and $D' \subseteq D$ we define

$$[A']R = \{<b, <c, d>> \in B\times(C \times D) \mid <a, b>R<c, d> \text{ for some } a \in A'\}$$
and $$R[D'] = \{<<a, b>, c> \in (A \times B)\times C \mid <a, b>R<c, d> \text{ for some } d \in D'\}.$$

Note that $[A']R[D']$ is unambiguous.

For $R_1 \subseteq A \times B$, $R_2 \subseteq B \times C$, and $A' \subseteq A$, it is easily shown

that $$[[A']R_1]R_2 = [A']R_1 \circ R_2 .$$

The latter expression is unambiguous since $([A']R_1)\circ R_2$ is meaningless. **We always omit parentheses when no ambiguity results.**

We also use the notion of a collection, which is a set with possibly repeated elements. ω^A denotes the set of all collections over set A . The union or 'sum' of collections μ and π we will denote by $\mu + \pi$. Note that $\mu + \mu \neq \mu$ when $\mu \neq \phi$:

$$\{a,b,a\} + \{a,a,b\} = \{a,a,b,a,b,a\} \quad \text{(order is unimportant).}$$

If Ψ is a set of collections, or asset of sets, $\cup\Psi$ **denotes** $\{a \mid a \in A \text{ for some } A \in \Psi\}$.

1: ASSERTION METHOD FOR A SIMPLE LANGUAGE

The programs in our Simple Language consist of sets of commands. Each command when executed changes the memory contents and causes control to transfer to another command. We will take the command as the most basic level of the definition. The language definition simply describes the effect of a program in terms of the basic operations of each command.

An arbitrary program P will consist of a set commands $C_P = \{c_1, c_2, \cdots, c_n\}$ and a set of labels $L_P = \{L_1, L_2, \cdots, L_n, L_{n+1}\}$. Each command c_i is labelled L_i. No command is labelled L_{n+1}. Command c_1 is called the initial command.

We assume a certain domain M_P of memory-contents. (We can think of the program as using a certain fixed number of variables, and each element of M_P is a vector of values for these variables.) The basic operation of each command c_i is then given by two functions $c_i^M : M_P \to M_P$ and $c_i^L : M_P \to L_P$. The first gives the new memory contents produced by executing c_i, and the second gives the label of the next command to be executed.

We now express the semantics of P in two ways.

A. First Method.

Let us call $Z_P = M_P \times L_P$ the set of states, $I_P = M_P \times \{L_1\}$ the set of initial states and $F_P = M_P \times \{L_{n+1}\}$ the set of final states. We shall express the semantics of P using a set of state-transformation rules derived from P, namely the rules

"$<m, \ell>$, where $\ell = L_i$, is transformed to $<c_i^M(m), c_i^L(m)>$"

for $1 \leq i \leq n$. These can more concisely be expressed as

$$(\forall m \in M_P) \langle m, L_1 \rangle \mapsto \langle c_i^M(m), c_i^L(m) \rangle \qquad \text{for } 1 \leq i \leq n .$$

A computation is a sequence of states $z_1, z_2, z_3, \cdots$ where $z_1 = \langle m_o, L_1 \rangle$ is an initial state, z_{j+1} is obtained from z_j $(j = 1,2,\cdots)$ by applying some transformation rule, and the sequence ends, if at all, when no rules are applicable, namely in a final state. The memory-contents in this final state are then said to be the <u>result</u> of executing P for <u>input</u> m_o .

B. <u>Second Method</u>.

Here we wish to express mathematically the relation $\underline{P}$ between inputs of P and the corresponding results.

For each $c_i \in C_P$, let $\underline{c_i} \subseteq Z_P \times Z_P$ be a relation defined by

$$\langle m, \ell \rangle \underline{c_i} \langle m', \ell' \rangle \text{ if and only if } \ell = L_i \;\; \& \;\; m' = c_i^M(m) \;\; \& \;\; \ell' = c_i^L(m) \; .$$

Let $\underline{C_P} = \bigcup_{i=1}^{n} \underline{c_i}$. Then $z \underline{C_P} z'$ whenever z and z' could be successive states in a computation.

Let $\underline{\text{yields}_P}$ be the reflexive transitive closure of $\underline{C_P}$. Then $z \; \underline{\text{yields}_P} \; z'$ whenever z and z' are the i-th and j-th states in some computation for some $i \leq j$.

The desired relation $\underline{P}$ between inputs and results is then $\{\langle m, m' \rangle \mid \langle m, L_1 \rangle \; \underline{\text{yields}_P} \; \langle m', L_{n+1} \rangle\}$ i.e. $\underline{P} = [L_1] \; \underline{\text{yields}_P} \; [L_{n+1}]$.

This is not a very 'efficient' definition of $\underline{P}$. As well as giving us $\underline{P}$, $\underline{\text{yields}_P}$ gives us the relations $[L_i]$ $\underline{\text{yields}_P}$ $[L_j]$ for all $L_i, L_j \in L_P$. It essentially specifies the relations between memory-contents computed between every pairs of commands in P . We can get better definitions by noting that

a) $[L_1]$ $\underline{\text{yields}_P}$ is the minimal relation $R \subseteq M_P \times Z_P$ satisfying both $R \circ \underline{C_P} \subseteq R$ and $E_{M_P} \subseteq R[L_1]$

b) $\underline{\text{yields}_P}$ $[L_{n+1}]$ is the minimal relation $R' \subseteq Z_P \times M_P$ satisfying both $\underline{C_P} \circ R' \subseteq R'$ and $E_{M_P} \subseteq [L_{n+1}]R'$.

$[L_1]$ $\underline{\text{yields}_P}$ and $\underline{\text{yields}_P}$ $[L_{n+1}]$ both specify fewer relations than $\underline{\text{yields}_P}$. (In fact $[L_1]$ $\underline{\text{yields}_P}$ gives us the relations between the input and the memory contents at each command, while $\underline{\text{yields}_P}$ $[L_{n+1}]$ gives us the relations between the memory contents at each command and the result of P .) And from both we can get $\underline{P}$ since

$\underline{P} = [L_1]$ $\underline{\text{yields}_P}$ $[L_{n+1}]$.

We now get corresponding versions of the assertion-method.

A. <u>First Method</u>.

Here we attach <u>assertions</u>, relations on M_P , to the labels in P , where the intention is that whenever a computation of P (for inputs satisfying the (<u>initial</u>) assertion attached to L_1) reaches a state $\langle m, L_i \rangle$ then the assertion attached to L_i must be true of the current memory-contents m . If this is the case we say the assertions are <u>valid</u> and then the (<u>final</u>) assertion attached to L_{n+1} express some property of the results of executing P .

We have to check that the attached assertions <u>are</u> valid, by ensuring that they satisfy a <u>verification-condition</u> related to the semantics of P . We shall be concerned here with correctly specifying the verification-condition itself, in order to ensure a sound and adequate assertion method.

We shall denote the attaching of assertions to P by a relation $A \subseteq M_P \times L_P$; for $L_i \in L_P$ the assertion (relation) attached to L_i is $A[L_i]$. We see that A is simply a relation on states. The assertions are <u>valid</u> if A is true of those states <u>yield</u>-ed from states in $A \cap I_P$, i.e. $[A \cap I_P]\ \underline{\text{yields}_P} \subseteq A$.

A <u>verification condition</u> V_P for P is then a relation on 2^{Z_P} . For the assertion-method using V_P to be sound we then require

I) (<u>Consistency of</u> V_P) $\forall A \subseteq Z_P : V_P(A) \Rightarrow [A \cap I_P]\ \underline{\text{yields}_P} \subseteq A$.

We also require the method to be adequate, that is, given valid initial and final assertions we should hope to be able to <u>prove</u> them valid using V_P :

II) (<u>Completeness of</u> V_P) $\forall A \subseteq I_P : V_P([A]\ \underline{\text{yields}_P})$

(To see this makes the method adequate note that if $\psi, \varphi \subseteq M_P$ are valid initial and final assertions for P then there exists an assignment of assertions, namely $[\psi \times \{L_1\}]\ \underline{\text{yields}_P}$, that satisfies V_P , and for which ψ implies the initial assertion and the final assertion implies φ .[*])

We can in fact specify such a complete and consistent verifi-

---

[*] To be logically complete, we should expect to be able to prove that these assertions satisfy V_P . In general this is impossible. These assertions are formulas of some calculus if and only if the calculus is incomplete.

cation condition for P :

Proposition 1.

If, for all $A \subseteq Z_P$, $V_P(A) \iff [A]\underline{C_P} \subseteq A$ then V_P is consistent and complete.

Proof.

a) Assume $V_P(A)$. Then $[A]\underline{C_P} \subseteq A$ giving

$[A]\underline{C_P} \circ \underline{C_P} = [[A]\underline{C_P}]\underline{C_P} \subseteq [A]\underline{C_P} \subseteq A$, and in general

$$[A]\underline{C_P}^{\textcircled{n}} \subseteq A \quad , \quad \text{for all } n \geq 1 \quad .$$

Since $A \subseteq A$ we get $[A]$ $\underline{\text{yields}_P} \subseteq A$ and hence $[A \cap I_P]$ $\underline{\text{yields}_P} \subseteq A$ and V_P is consistent.

b) Since $\underline{\text{yields}_P} \circ \underline{C_P} \subseteq \underline{\text{yields}_P}$ by definition, for all $A \subseteq I_P$, $[[A]\ \underline{\text{yields}_P}]\underline{C_P} = [A]\ \underline{\text{yields}_P} \circ \underline{C_P} \subseteq [A]\ \underline{\text{yields}_P}$. Thus $V_P([A]\ \underline{\text{yields}_P})$ and V_P is complete.

$\square$

By substituting the definition of $\underline{C_P}$ and simplifying it is easy to show that for all $A \subseteq Z_P$

$$V_P(A) \iff \forall m \in M_P \mathop{\&}_{c_j \in C_P} A(m, L_j) \Rightarrow A(c_j^M(m), c_j^L(m)) \quad .$$

In more concrete versions of the Simple Language, the functions c_j^L are finitely expressible as for example $c_i^L(m) = \underline{\text{if}}\ p_i(m)\ \underline{\text{then}}\ L_j$ $\underline{\text{else}}\ L_k$. In such cases we can express V_P as a formula of predicate calculus:

$$V_P = \forall m \bigwedge_{c_j \in C_P} q_{L_j}(m) \supset q_{c_j^L(m)}(c_j^M(m))$$

where the term in the conjuction for the above example would in fact be

$$q_{L_i}(m) \supset \underline{\text{if}}\ p_i(m)\ \underline{\text{then}}\ q_{L_j}(c_i{}^M(m))\ \underline{\text{else}}\ q_{L_k}(c_i{}^M(m)) .$$

To check $V_P(A)$ we then simply interpret each predicate symbol q_{L_i} as $A[L_i]$, the assertion attached to L_i, and check if the formula is satisfied. If it is, then the assertions are valid and $A[L_{n+1}]$ is true of the results of computations of P for inputs satisfying $A[L_1]$.

This is the formula and method of Manna [10], based on the earlier work of Floyd [5].

B. Second Method.

Here we associate assertions with the relations used to specify the input-results relation $\underline{P}$. We require that the assertions be true of those values which satisfy the relations, and hence be true of the inputs and results of P. There were two ways of specifying $\underline{P}$.

a) Using $[L_1]\ \underline{\text{yields}_P}$. Here we associate assertions by giving a relation $A \subseteq M_P \times Z_P$. $A[L_i]$ will be the assertion associated with the relation $[L_1]\ \underline{\text{yields}_P}\ [L_i]$ and A will be valid if $[L_1]\ \underline{\text{yields}_P} \subseteq A$.

b) Using $\underline{\text{yields}_P}\ [L_{n+1}]$. Here we take $A \subseteq Z_P \times M_P$. $[L_i]A$ will be the assertion associated with the relation $[L_i]\ \underline{\text{yields}_P}\ [L_{n+1}]$ and A will be valid if $\underline{\text{yields}_P}\ [L_{n+1}] \subseteq A$.

A verification condition V_P for P will then be a relation on $2^{M_P \times Z_P}$ in case a) and on $2^{Z_P \times M_P}$ in case b) . In the two cases we get the following conditions for the soundness and adequacy of the assertion method using V_P :

I) (Consistency of V_P)

a) for all $A \subseteq M_P \times Z_P : V_P(A) \Rightarrow [L_1]\ \underline{\text{yields}}_P \subseteq A$

b) for all $A \subseteq Z_P \times M_P : V_P(A) \Rightarrow \underline{\text{yields}}_P\ [L_{n+1}] \subseteq A$

II) (Completeness of V_P)

a) $V_P([L_1]\ \underline{\text{yields}}_P)$

b) $V_P(\underline{\text{yields}}_P\ [L_{n+1}])$.

It is obvious that we have ready-made verification-conditions V_P in the two cases - we simply use the defining conditions for $[L_1]\ \underline{\text{yields}}_P$ and $\underline{\text{yields}}_P\ [L_{n+1}]$ respectively:

a) For all $A \subseteq M_P \times Z_P$

$$V_P(A) \Leftrightarrow A \circ \underline{C}_P \subseteq A \quad \& \quad E_{M_P} \stackrel{L}{\subseteq} A(L_1) \ .$$

To get a predicate calculus version of V_P we first substitute for $\underline{C}_P$ and get

$$V_P(A) \Leftrightarrow \forall m_o, m \in M_P : mA\langle m, L_1\rangle \quad \& \quad \mathop{\&}_{c_i \in C_P} m_oA\langle m, L_i\rangle \Rightarrow m_oA\langle c_i^M(m), c_i^L(m)\rangle.$$

Then writing V_P as a formula of predicate calculus, we get

$$V_P = (\forall m_o, m) q_{L_1}(m, m) \wedge \bigwedge_{c_i \in C_P} q_{L_i}(m_o, m) \supset q_{c_i{}^L(m)}(m_o, c_i^M(m))$$

where the term in the conjunction for the example in the first method would in fact be

$$q_{L_i}(m_o, m) \supset \underline{\text{if}}\ p_i(m)\ \underline{\text{then}}\ q_{L_j}(m_o, c_i^M(m))\ \underline{\text{else}}\ q_{L_k}(m_o, c_i^M(m))$$

This is the formula used in Manna [11].

b) For all $A \subseteq Z_P \times M_P$

$$V_P(A) \iff \underline{C}_P \circ A \subseteq A \ \ \& \ \ E_{M_P} \subseteq [L_{n+1}]A \ .$$

To get a predicate calculus version of V_P we first substitute for $\underline{C}_P$ and get

$$V_P(A) \iff \forall m,m' \in M_P : \langle m, L_{n+1}\rangle A m \ \ \& \ \ \mathop{\&}_{c_i \in C_P} \langle c_i^M(m), c_i^L(m)\rangle A m' \Rightarrow \langle m, L_i\rangle A\, m' \ .$$

As a formula of predicate calculus V_P becomes

$$V_P = (\forall m,m')\, Q_{L_{n+1}}(m, m) \ \ \& \bigwedge_{c_i \in C_P} Q_{c_i^L(m)}(c_i^M(m), m') \supset Q_{L_i}(m, m')$$

where the term in the conjunction in the previous example would in fact be

$$[\underline{\text{if}}\ p_i(m)\ \underline{\text{then}}\ Q_{L_j}(c_i^M(m), m')\ \underline{\text{else}}\ Q_{L_k}(c_i^M(m), m')] \supset Q_{L_i}(m, m')$$

This is the formula that would be obtained by first converting P to a recursive program using the technique of McCarthy [14], and then using the method of Manna and Pnueli [13].

To check $V_P(A)$, in case a) we interpret predicate symbol q_{L_i} as $A[L_i]$ and in case b) we interpret predicate symbol Q_{L_i} as $[L_i]A$. If V_P is then satisfied the assertions are valid: in case a) $A[L_i]$ extends the relation (between memory-contents) between the input of P and label L_i ; in case b) $[L_i]A$ extends the relation (between memory-contents) between label L_i and the results of P . $A[L_{n+1}]$ and $[L_1]A$ respectively in the two cases extend $\underline{P}$ and are thus true of the inputs and corresponding results of P .

Examples of the use of these type of formulae for proving properties of programs can be found in the references.

We have seen for this Simple Language that the assertion-method, in its theoretical justification and in its application, is closely linked to the type and form of language definition used. To explore this relationship more fully we first consider language definition in more detail.

2: METHODS OF LANGUAGE DEFINITION

There are two basic philosophical approaches to the specification of programming language semantics.

The first approach, the operational, computational or intensional approach, gives meaning to programs by describing the ways in which their computations proceed. This method enables one to find the results of a program for any inputs, but is more concerned with describing the manner in which these results are obtained.

The second approach, the denotational or extensional approach, is concerned only with specifying the inputs-results relationships of programs. These relationships are given mathematically and the mathematical specification need not suggest the actual mechanism for obtaining the results for given inputs. In fact the specification may be non-constructive.

Both methods have their advantages for different applications. The first method is more useful to implementers and programmers, while the second is preferable for proving things about the results of programs, and is aesthetically more satisfying.

Computational Definitions.

By and large computational definitions fall into two classes: interpreter definitions (for example [9 ,15]) and transition-rule definitions (for example [4]). In the first case an abstract interpreter is given for the programming language, written in some suitably formalised algorithmic language. Any program is then considered as data, as is the programs input, and the execution of the interpreter describes the computation of the program for this input. In the second case we consider data structures called 'states' and the computations of programs are given as sequences of states produced by application of appropriate 'state-transition rules'. This class can be further subdivided depending on the role played by the programs of the language. We can either incorporate the programs into the state and have one set of rules for the language itself, or we can embed the programs in the rules, and use simpler states. The latter method requires a translator from programs to transition rules, which can be thought of as a sort of compiler into the basic 'transition-rule language'. (The former method usually also requires a translation of programs into their 'state representation'.)

The distinction between the classes is really only one of degree; the interpreter definitions usually talk of states also, and conversely the process of repeatedly applying transition rules can be thought of as the behaviour of a simple one-loop interpreter. Nevertheless we shall keep the distinction, and prefer definitions in the second class because of their simplicity and similarities to other formal systems, such as formal grammars and predicate calculus. In fact we shall restrict ourselves to considering only the transition-rule - compiler definitions, to simplify the 'states'. Our aim is to obtain sound and adequate generalisations of the assertion-method for languages defined in this way. (Notice that the first definition of the Simple Language was a transition-rule-compiler definitions)

At any stage in a computation the state must contain all the data that will be needed in the rest of the computation. In the Simple Language, with a fixed number of variables and simple control structure, this amount of data is also fixed; we need only the memory contents and the 'instruction counter'. For languages with recursion however, states must hold an unbounded amount of data. This is inconvenient when applying the assertion-method; it is difficult to assert properties of such states. It is preferable to have states containing fixed amounts of data, any item of which can be referred to directly.

The solution is to allow several 'states' to exist at once, in fact we allow an unbounded number of them. Each 'state' will then correspond to some part or aspect of the previous monolithic state, and we shall call them 'situations' rather than states. The previous state-transition rules would only have made localised changes in the state, and these changes are now described by changes in a subset of the current situations, the rest of the situations staying the same. Our aim in introducing 'situations' is that it should be natural to express their properties , so the situations themselves should describe 'meaningful' aspects of the state. The transition rules then should become directly intelligible (and even the idea of the 'state' can become unnecessary).

We can formalise these ideas as follows. To give the semantics of some language we need to specify for each program P a domain of <u>situations</u> S_P and a set of transition rules $R_P = \{R_1, R_2, \cdots, R_n\}$. Each rule R_i denotes a relation $\underline{R_i} \subseteq 2^{S_P} \times 2^{S_P}$. The idea is that for $A, B \subseteq S_P$, if $A \underline{R_i} B$ and the current state (<u>collection</u> of and $f(n_1) = n_2$ meaning "f called with n_1 returns n_2 " .

The initial situations are those in the first class.

The rules R_P are as follows:

R_1 : $(\forall x \in J)P(x) \longmapsto P(x), f(x)$

R_2 : $(\forall x,y \in J)P(x), f(x) = y \longmapsto P(x) = y$

R_3 : $(\forall x \in J, x > 100) f(x) \longmapsto f(x) = x - 10$

R_4 : $(\forall x \in J, x \leq 100) f(x) \longmapsto f(x), f(x + 11)$

R_5 : $(\forall x,y \in J, x \leq 100) f(x), f(x + 11) = y \longmapsto f(x), f(x + 11) = y, f(y)$

R_6 : $(\forall x,y,z \in J, x \leq 100) f(x), f(x + 11) = y, f(y) = z \longmapsto f(x) = z$.

These rules denote relations in an obvious way. For example, for $A,B \subseteq S_P$, $A \, \underline{R_5} \, B$ if and only if for some $n_1,n_2 \in J$, $n_1 \leq 100$ and

$$A = \{f(n_1), f(n_1 + 11) = n_2\}$$
$$B = \{f(n_1), f(n_1 + 11) = n_2, f(n_2)\} .$$

The computation of P for initial situation P(99) , for example, is then

$\{P(99)\}$

$\{P(99), f(99)\}$

$\{P(99), f(99), f(110)\}$

$\{P(99), f(99), f(110) = 100\}$

$\{P(99), f(99), f(110) = 100, f(100)\}$

{P(99), f(99), f(110) = 100, f(100), f(111)}

{P(99), f(99), f(110) = 100, f(100), f(111) = 101}

{P(99), f(99), f(110) = 100, f(100), f(111) = 101, f(101)}

{P(99), f(99), f(110) = 100, f(100), f(111) = 101, f(101) = 91}

{P(99), f(99), f(110) = 100, f(100) = 91}

{P(99), f(99) = 91}

{P(99) = 91}

(In this particular computation each state is a *set* of situations, but for computations from initial situations P(x) where $x < 90$ we will in fact get states containing repeated situations.)

This particular program P has rules R_P with a very important property. Let $\underline{\text{yields}}_P$ be the reflexive transitive closure of $\underline{\underline{R}}_P$. Then

(*The independence property*)

for all $A \subseteq S_P$ and $\Phi \subseteq \Sigma_O$, if $A \subseteq \bigcup[\Phi]$ $\underline{\text{yields}}_P$ and $[A]\underline{\underline{R}}_P \neq \phi$ then for some $\sigma \in \Sigma_P$ containing A and some $\sigma_O \in \Phi$ σ_O $\underline{\text{yields}}_P$ σ .

This property says that if the various situations that would allow a rule to be applied occur at all at any time during any of the computations of P from initial states Φ then there must be a computation, from some *single* initial state in Φ , in which they all occur *simultaneously*.

We will see that this property allows us to derive verification conditions very directly from the situation-transition rules for programs. We shall therefore restrict ourselves to definitions with the independence property. This is not a very strong restriction; languages involving recursion seem to require definitions with this property anyway.

We used the notion of 'state' to formalise the idea of a computation. One advantage of 'situational' definitions however is that the rules do not refer to states. The rules describe the semantics of the various language constructs using the minimal amount of 'environment'. This makes the rules very intelligible as intuitive descriptions of the constructs. The rules for the 91-function for example could be written in English with very little change, and the result is a succinct but understandable description of the way the program computes.

For a situational definition of most of Algol 60 see [3].

Denotational Definitions.

The denotation method associates with each program in a language a set of partial functions or relations. This set contains the partial function or relation "computed by the program", i.e. the inputs-results function or relation. These functions or relations are specified for each program as the 'least-defined' or minimal ones satisfying conditions derived from the program itself; the semantics of the language can be considered to be embodied in this translation of programs into conditions. This was the method used in the second definition of the Simple Language.

To some extent the functional and relational specifications are interchangeable. The relations will turn out to be graphs of functions, and then 'minimal' relation is the same thing as 'least-defined' function. However there can be differences. If a function is not strict in some its arguments (i.e. if it can give a 'defined' result when one of its arguments is undefined) then it doesn't have a graph in the usual sense, unless we introduce a special element to mean 'undefined'. And if we do this then 'least-defined' function no longer means 'minimal' graph, in a set-theoretic sense. The relational approach essentially restricts us to the graphs of strict functions.

We are going to only consider in future the relational approach (or functional approach limited to strict functions) because of the simplicity of adapting the assertion method to languages defined in this way. This is a restriction because there are languages for which a natural denotational definition must allow non-strict functions, for example recursive programs with 'call-by-name' type evaluation rules (see [12]).*) (It is interesting that the situational approach, which is also closely related to the assertion-method, also seems unable to handle 'call-by-name'). Nevertheless we feel that many interesting languages

*) In [12] it is argued that 'call-by-value' evaluation rules are incorrect since they are not 'fixed point rules'. However if we restrict ourselves to strict functions, and insist that all expressions in programs be strict, the 'minimal fixed points' are given by 'call-by-value' evaluation rules. Whether we choose minimal minimal fixed points (call-by-value) or maximal minimal fixed points (call-by-name) is largely a matter of taste. I am indebted to David Park for this observation.

can be defined this way (see [6]).

The conditions obtained in the functional approach usually consist of an k-functional τ (map from k-tuples of functions to k-tuples of functions) of which we require the minimal fixed point, i.e. the k-tuple of least defined functions $\bar{f}$ such that $\tau(\bar{f}) = \bar{f}$. In fact this is the same as asking for the least defined $\bar{f}$ such that $\tau(\bar{f})$ is no more defined than $\bar{f}$. In relational terms, we are looking for the (set-theoretically) minimal k-tuple of relations $\bar{r}$ such that $\tau'(\bar{r}) \subseteq \bar{r}$ (where τ' is the translation of τ into relational terms).

In a typical denotational definition then,we specify the semantics of a program as a set of relations, defined as the minimal relations satisfying some inclusion (or set of inclusions). The definition of the language is a translation of programs into inclusions. Our previous denotational definition of the Simple Language was of this type.

Example: As an example we can consider again the 91-function:

$$f(x) \Leftarrow \underline{\text{if}}\ x > 100\ \underline{\text{then}}\ x - 10\ \underline{\text{else}}\ f(f(x + 11)) .$$

We specify the semantics of f as the least defined (strict) function F satisfying

$$F(x) = \underline{\text{if}}\ x > 100\ \underline{\text{then}}\ x - 10\ \underline{\text{else}}\ F(F(x + 11)) .$$

In relational terms, we want the minimal binary relation R (over integers) satisfying $\tau'(R) \subseteq R$, where for $x,y \in J$

$$\tau'(R)(x, y) \iff (x > 100 \ \& \ y = x - 10) \lor$$
$$(x \leq 100 \ \& \ \exists z \in J[(x + 11) R z \ \& \ z R y])$$ [*]

A denotational definition of the language of recursive programs would specify an appropriate τ' for each program.

[*] Hitchcock & Park [6] have a relational calculus for expressing such definitions which avoids the use of individual variables.

3: THE ASSERTION METHOD

The way we associate or attach assertions to a program, and the meaning of these assertions depends on the way in which the program's semantics are specified. This is because the criteria for the correctness of a particular version of the assertion method (apapted to a particular language) depend on the form of the semantic definition of the language. The two forms we will consider are situational definitions, and relational denotational definitions.

Situational Definitions.

The assertions attached to a program P are intended to be true of the situations that actually occur in computations of P (for initial situations satisfying the 'initial assertion'). Such assertions are said to be valid.

We shall take the attached assertions as simply a relation $A \subseteq S_P$ on situations; the 'initial assertion' is $A \cap S_O$. (We assume the notation of the previous section.)

For a sound and adequate assertion-method for P we then require a verification condition $V_P \subseteq 2^{S_P}$ with the following properties:

I) (Consistency of V_P) For all $A \subseteq S_P$

$$V_P(A) \Rightarrow \cup[(A \cap S_O)^1] \underline{\text{yields}_P} \subseteq A$$

(i.e. the assertions are valid)

II) (Completeness of V_P) For all $\Phi \subseteq \Sigma_O$, $V_P(\cup[\Phi]\ \underline{\text{yields}_P})$.

<u>Obtaining</u> V_P <u>from the rules</u> R_P .

First we establish some useful properties of $\underline{\underline{R}}_P$ and <u>yields</u>$_P$.

<u>Property 1</u>.

For all $\Theta \subseteq \Sigma_P$

$$\bigcup[\Theta]\underline{\underline{R}}_P \subseteq \bigcup[\bigcup\Theta]\underline{\underline{R}}_P \cup \bigcup\Theta \ .$$

<u>Proof</u>. Consider any situation $s \in \bigcup[\Theta]\underline{\underline{R}}_P$. By definition of $\underline{\underline{R}}_P$ there exist states $\sigma_1,\sigma_2 \in \Sigma_P$, and $B,C \subseteq S_P$ such that $\sigma_1 \in \Theta$, $\sigma_1 = \sigma_2 + B$, $B\ \underline{\underline{R}}_P\ C$ and s is contained in $\sigma_2 + C$.

If s is contained in σ_2 then s is contained in σ_1 .

$$\therefore\ s \in \bigcup\Theta \ .$$

If s is contained in C , since $B \subseteq \bigcup\Theta$ and $B\ \underline{\underline{R}}_P\ C$,

$$s \in \bigcup[\bigcup\Theta]\underline{\underline{R}}_P \ .$$

□

<u>Property 2</u>.

For $A,B \subseteq S_P$, if $A \subseteq B$ then

$$\bigcup[A]\underline{\underline{R}}_P \subseteq \bigcup[B]\underline{\underline{R}}_P \ .$$

<u>Proof</u>.

Immediate from the definition of $\underline{\underline{R}}_P$.

□

Property 3.

If we have the independence property then for all $\Phi \subseteq \Sigma_0$

$$\bigcup[\bigcup[\Phi]\underline{\text{yields}}_P]\underline{\underline{R}}_P \subseteq \bigcup[\Phi]\underline{\text{yields}}_P \,.$$

Proof.

Consider any situation $s \in \bigcup[\bigcup[\Phi]\underline{\text{yields}}_P]\underline{\underline{R}}_P$. By definition of $\underline{\underline{R}}_P$ there exists $A,B \subseteq S_P$ such that $A \subseteq \bigcup[\Phi]\underline{\text{yields}}_P$, $A\ \underline{\underline{R}}_P\ B$ and s is contained in $B + (\bigcup[\Phi]\underline{\text{yields}}_P - A)$.

If $s \in B$ then, since $[A]\underline{\underline{R}}_P \neq \phi$, by the independence property there exists some $\sigma_0 \in \Phi$ such that $\sigma_0\ \underline{\text{yields}}_P\ \sigma$ for some $\sigma \in \Sigma_P$ containing A . Thus $s \in \bigcup[\sigma_0]\underline{\text{yields}}_P \circ \underline{\underline{R}}_P$ and

$$s \in \bigcup[\Phi]\underline{\text{yields}}_P \,.$$

If $s \notin B$ then $s \in \bigcup[\Phi]\underline{\text{yields}}_P$ directly.

$\square$

Property 4.

For all $I \subseteq S_0$

$$\bigcup[I^1]\underline{\text{yields}}_P \subseteq \bigcup[I]\underline{\text{yields}}_P$$

Proof.

It is straightforward by induction on n that, for all $n \geq 0$, if state $\sigma \in [I^1]\underline{\underline{R}}_P^{\textcircled{n}}$ then $I\ \underline{\text{yields}}_P\ \sigma_1$ for some $\sigma_1, \sigma_2 \in \Sigma_P$ such that $\sigma_1 = \sigma + \sigma_2$. This immediately proves the property.

$\square$

For completeness we also prove

<u>Property 5.</u>

If we have the independence property then for all $I \subseteq S_0$

$$\bigcup[I]\underline{\text{yields}}_P \subseteq \bigcup[I^1]\underline{\text{yields}}_P \ .$$

<u>Proof.</u>

We prove that for all $n \geq 0$

$$\bigcup[I]\underline{\underline{R}}_P^{\textcircled{n}} \subseteq \bigcup[I^1]\underline{\text{yields}}_P \ . \qquad (1)$$

<u>n = 0</u> $\quad \bigcup\{I\} = I = \bigcup I^1 \subseteq \bigcup[I^1]\underline{\text{yields}}_P \ .$

Assume (1) is true for $n = k$.

<u>n = k + 1</u>

$$\begin{aligned} \bigcup[I]\underline{\underline{R}}_P^{\textcircled{n}} &= \bigcup[[I]\underline{\underline{R}}_P^{\textcircled{k}}]\underline{\underline{R}}_P \\ &\subseteq \bigcup[\bigcup[I]\underline{\underline{R}}_P^{\textcircled{k}}]\underline{\underline{R}}_P \cup \bigcup[I]\underline{\underline{R}}_P^{\textcircled{k}} \end{aligned}$$

by Property 1.

Hence by the Induction Hypothesis, and Property 2

$$\bigcup[I]\underline{\underline{R}}_P^{\textcircled{n}} \subseteq \bigcup[\bigcup[I^1]\underline{\text{yields}}_P]\underline{\underline{R}}_P \cup \bigcup[I^1]\underline{\text{yields}}_P \ .$$

By Property 3

$$\bigcup[\bigcup[I^1]\underline{\text{yields}}_P]\underline{\underline{R}}_P \subseteq \bigcup[I^1]\underline{\text{yields}}_P \ .$$

$$\therefore \bigcup[I]\underline{\underline{R}}_P^{\textcircled{n}} \subseteq \bigcup[I^1]\underline{\text{yields}}_P \ .$$

$\square$

Corollary 1.

If we have the independence property then, for all $I \subseteq S_o$, $\cup[I]\underline{\text{yields}}_P = \cup[I^1]\underline{\text{yields}}_P$.

(This is the motivation for calling it the independence property. The corollary essentially says that if we run several computations together we get the same situations as if we run the computations separately. The computations are independent.)

Corollary 2.

Property 3 is true also for all $I \subseteq S_o$. (In place of $\Phi \subseteq \Sigma_o$)

We can now get a particular formulation of V_P.

Proposition 2.

If we have the independence property and, for all $A \subseteq S_P$, $V_P(A) \iff \cup[A]\underline{\underline{R}}_P \subseteq A$ then V_P is consistent and complete.

Proof.

a) Assume $V_P(A)$. Then $\cup[A]\underline{\underline{R}}_P \subseteq A$ and thus $\cup[\cup[A]\underline{\underline{R}}_P]\underline{\underline{R}}_P \subseteq A$.

Now by property 1

$$\cup[A]\underline{\underline{R}}_P^{\textcircled{2}} = \cup[[A]\underline{\underline{R}}_P]\underline{\underline{R}}_P \subseteq \cup[\cup[A]\underline{\underline{R}}_P]\underline{\underline{R}}_P \cup \cup[A]\underline{\underline{R}}_P \subseteq A .$$

We can repeat the process indefinitely and get

$$\cup[A]\underline{\underline{R}}_P^{\textcircled{n}} \subseteq A \qquad \text{for all } n \geq 1 .$$

Since $\cup\{A\} = A$ we get $\cup[A]\,\underline{\text{yields}}_P \subseteq A$ and by Property 2 $\cup[A \cap S_o]\,\underline{\text{yields}}_P \subseteq A$. Then by Property 4, $\cup[(A \cap S_o)^1]\,\underline{\text{yields}} \subseteq A$ and V_P is consistent.

b) By Property 3 we immediately have that V_P is complete.

□

We can express this formulation of V_P more directly in terms of the rules R_P . Note that the definitions of $\underline{\underline{R}}_P$ and $\underline{R}_P$ imply that for all $A \subseteq S_P$

$$\begin{aligned}\cup[A]\underline{\underline{R}}_P \subseteq A \quad &\Longleftrightarrow \quad \forall B \subseteq A, \forall C \subseteq S_P,\ B\ \underline{\underline{R}}_P\ C \Rightarrow C \subseteq A \\ &\Longleftrightarrow \quad \forall B,C \subseteq S_P,\ B\ \underline{R}_P\ C \text{ implies } (B \subseteq A \Rightarrow C \subseteq A) \\ &\Longleftrightarrow \quad \forall R_i \in R_P, \forall B,C \subseteq S_P,\ B\ \underline{R}_i\ C \text{ implies} \\ &\qquad\qquad (B \subseteq A \Rightarrow C \subseteq A)\end{aligned}$$

Thus, for program P , defined by rules R_P with the independence property, a complete and consistent verification condition for checking validity of assertions $A \subseteq S_P$ is the following:

$\forall R_i \in R_P$, $\forall B,C \in S_P$, $B\ \underline{R}_i\ C$ implies $(B \subseteq A \Rightarrow C \subseteq A)$

i.e. for all rules, if the situations on the left of the '$\mapsto$' satisfy the assertions A then the situations on the right must also satisfy A .

This particularly simple generalisation of the assertion method is the main justification for developing the 'situational' definition technique.

Example: the 91-function.

We shall illustrate the assertion method by proving the validity of the following assertions A :

$$\begin{aligned} \forall x,y \in J \qquad & A(P(x)) \iff x \le 101 \\ & A(P(x) = y) \iff y = 91 \\ & A(f(x)) \iff x \le 111 \\ & A(f(x) = y) \iff y = \max(x-10, 91) \quad . \end{aligned}$$

Simply applying the verification condition, for the first rule R_1 we have to check that

$$(\forall x \in J) A(P(x)) \Rightarrow A(P(x)) \; \& \; A(f(x))$$

i.e. (1) $(\forall x \in J) \; x \le 101 \Rightarrow x \le 111$

(we drop the repeated term).

Similarly for the other rules we have the check

(2) $(\forall x,y \in J) \quad x \le 101 \;\; \& \;\; y = \max(x-10, 91) \Rightarrow y = 91$

(3) $(\forall x \in J) \quad x \le 111 \;\; \& \;\; x > 100 \Rightarrow x - 10 = \max(x-10, 91)$

(4) $(\forall x \in J) \quad x \le 111 \;\; \& \;\; x \le 100 \Rightarrow x + 11 \le 111$

(5) $(\forall x,y \in J) \quad x \le 111 \;\; \& \;\; x \le 100 \;\; \& \;\; y = \max(x+1, 91) \Rightarrow y \le 111$

(6) $(\forall x,y \in J) \quad x \le 111 \;\; \& \;\; x \le 100 \;\; \& \;\; y = \max(x+1, 91) \;\; \&$

$z = \max(y, 91) \Rightarrow z = \max(x, 91) \quad .$

All these are trivial properties of integers. We have thus proved that for all computations of P for integer inputs less than 102, if we ever get the situation $P(x) = y$, then $y = 91$. (Situations of this type would be distinguished as _final_ assertions.) Note also that if we ever get the situation $f(x)$ then $x \leq 111$. Thus if we run P for inputs less than 102 we never call f for inputs greater than 111. This is a typical property of the _computations_ of P (as distinct from the _results_ of P) that is provable by this method.

Expressing V_P as a Predicate Calculus Formula.

When the situations S_P fall into a finite number of disjoint classes $S_0, S_1, \cdots, S_m$ and each situation in a given class S_i contains a fixed number n_i of data items (as in this example), we can express V_P as a formula of predicate calculus. Any situation in S_i can be represented as $\langle\bar{x}\rangle_i$, where $\bar{x}$ is a vector of data items of length n_i; the subscript i is simply notation to denote the class or type of the situation. To specify assertions $A \subseteq S_P$ we then have to specify $A(\langle\bar{x}\rangle_i)$ for all n_i-tuples of data items $\bar{x}$, and $1 \leq i \leq m$. The assertions A then become essentially m relations A_i on _n_i-tuples of data items_. In the example the assertions would be

$$A_0(x) \iff x \leq 101$$

$$A_1(x, y) \iff y = 91$$

$$A_2(x) \iff x \leq 111$$

$$A_3(x, y) \iff y = \max(x-10, 91) .$$

We can then express V_P using n_i-ary predicate symbols q_i which we interpret as relations A_i whenever we want $V_P(A)$. In the example V_P becomes

$$\begin{aligned}
\forall(x,y,z \in J)[[&q_o(x) \supset q_1(x)] \\
\& \ [&q_o(x) \ \& \ q_3(x, y) \supset q_1(x, y)] \\
\& \ [&q_1(x) \ \& \ x > 100 \supset q_2(x, x-10)] \\
\& \ [&q_1(x) \ \& \ x \leq 100 \supset q_1(x + 11)] \\
\& \ [&q_1(x) \ \& \ x \leq 100 \ \& \ q_2(x+11, y) \supset q_1(y)] \\
\& \ [&q_1(x) \ \& \ x \leq 100 \ \& \ q_2(x+11, y) \ \& \ q_2(y, z) \\
&\qquad\qquad \supset q_2(x, z)]]
\end{aligned}$$

The construction of this formula directly from the rules R_P is obvious.

Formulae of this type, for recursive programs, would be obtained using the technique of Ashcroft [1].

Relational Denotational Definitions.

We will consider two versions of the assertion method for languages defined denotationally using relations. These are, respectively, the methods of Park and Hoare.

Park's Method:

As for the Simple Language we associate assertions with the relations specified by the definition of the program. Each assertion is itself a relation, and to be valid it must simply extend the corresponding defined relation. Hence the inputs and results of the program must satisfy the relation (assertion) associated with the defined inputs-results relation.

Assume the semantics of program P are specified as a k-tuple of relations $\bar{r}$ (for some k) which are the minimal relations such that $\tau_P(\bar{r}) \subseteq \bar{r}$ (where this can be considered as a k-tuple of inclusions). Assertions $\bar{A}$ 'attached' to P now also consist of a k-tuple of relations.

For a sound and adequate assertion method we require a verification V_P (a relation on k-tuples of relations) such that

I) (Consistency of V_P) For all k-tuples of relations $\bar{A}$

$$V_P(\bar{A}) \Rightarrow \bar{r} \subseteq \bar{A}$$

II) (Completeness of V_P) $V_P(\bar{r})$

<u>Obtaining V_P from the definition of P</u>

Clearly, as for the Simple Language, we can take $V_P(\bar{A}) \iff \tau_P(\bar{A}) \subseteq \bar{A}$. By definition $\bar{r}$ is the minimal k-tuple of relations satisfying V_P and I) and II) are guaranteed.

<u>Example: the 91-function</u>

From the definition of the 91-function given earlier we see $\bar{A}$ is a single binary relation A_1 on integers and

$$V_P(\bar{A}) \iff (\forall x,y \in J)$$
$$[(x > 100 \;\&\; y = x - 10)$$
$$\vee\; x \leq 100 \;\&\; (\exists z \in J)[(x + 11)A_1 z \;\&\; z\, A_1 y]] \Rightarrow x\, A_1 y$$

Now if we define A_1 as

$$x A_1 y \iff (x > 100 \;\&\; y = x - 10) \vee (x \leq 100 \;\&\; y = 91)$$

to check $\bar{A}$ ($\equiv A_1$) for validity we have to check

$$\begin{aligned}
&\forall x,y \in J \\
&\quad [(x > 100 \;\&\; y = x - 10) \vee \\
&\quad\; x \leq 100 \;\&\; (\exists z \in J)[((x + 11 > 100 \;\&\; z = x + 1) \vee \\
&\quad\; (x + 11 \leq 100 \;\&\; z = 91)) \;\&\; ((z > 100 \;\&\; y = z - 10) \vee \\
&\quad\; (z \leq 100 \;\&\; y = 91))]] \\
&\quad \Rightarrow (x > 100 \;\&\; y = x - 10) \vee (x \leq 100 \;\&\; y = 91) .
\end{aligned}$$

This breaks up into the following five statements:

$\forall x,y,z \in J$

(i) $(x > 100 \;\&\; y = x - 10) \Rightarrow (x > 100 \;\&\; y = x - 10) \vee (x \leq 100 \;\&\; y = 91)$

(ii) $x \leq 100 \;\&\; x + 11 > 100 \;\&\; z = x + 1 \;\&\; z > 100 \;\&\; y = z - 10$
$\Rightarrow (x > 100 \;\&\; y = x - 10) \vee (x \leq 100 \;\&\; y = 91)$

(iii) $x \leq 100 \;\&\; x + 11 > 100 \;\&\; z = x + 1 \;\&\; z \leq 100 \;\&\; y = 91$
$\Rightarrow (x > 100 \;\&\; y = x - 10) \vee (x \leq 100 \;\&\; y = 91)$

(iv) $x \leq 100 \;\&\; x + 11 \leq 100 \;\&\; z = 91 \;\&\; z > 100 \;\&\; y = z - 10$
$\Rightarrow (x > 100 \;\&\; y = x - 10) \vee (x \leq 100 \;\&\; y = 91)$

(v) $x \leq 100 \;\&\; x + 11 \leq 100 \;\&\; z = 91 \;\&\; z \leq 100 \;\&\; y = 91$
$\Rightarrow (x > 100 \;\&\; y = x - 10) \vee (x \leq 100 \;\&\; y = 91)$.

All these statements are trivial properties of integers. Hence A_1 is a valid relation. Hence, in particular, if the input to the 91-function is no more than 101 then the result, if any, is 91. This we also proved from the situational definition and assertion method. However in the latter proof we also proved that for inputs no more than 101 we never called f for values greater than 111. This we can not prove from the denotational definition - the concept of 'calling a function' is not defined.

<u>Expressing V_P as a predicate calculus formula</u>

It is clear that once again we can express V_P as a formula of predicate calculus - we simple replace the k relations $\bar{A}$ by k predicate symbols $Q_1, Q_2, \cdots, Q_k$ in $\tau_P(\bar{A}) \subseteq \bar{A}$. Thus for the 91-function V_P becomes (after a little manipulation)

$$(\forall x,y,z \in J)\,[x > 100 \Rightarrow Q_1(x,\ x-10)]$$
$$\& \quad [x \leq 100 \ \& \ Q_1(x+11,\ z) \ \& \ Q_1(z,\ y) \Rightarrow Q_1(x,\ y)]$$

To check $V_P(\bar{A})$ we simply interpret the predicate symbol Q_1 as the relation A_1.

Formulae of this type, for recursive programs, would be obtained by the technique of Manna & Pnueli [13]. It is interesting that their formulae were justified using a <u>computational</u> definition of recursive programs. The more elegant justification, using denotational definitions, is due to Park [16]. It is also worth noting that the author's technique [1] for recursive programs was developed independently of Manna and Pnueli, and the differences in the two techniques come solely from the differences between the two approaches to language definition, computational and denotational.

Hoare's Method.

At first sight Hoare's method [7 , 8] of proving assertions about programs appears to be a direct application of Floyd's method, and hence appears based on computational semantic definitions of programs. Assertions are attached to programs in the same way, between statements, and the aim of the method is to show that they are true whenever computation reaches them.

However, we will show that there is an implicit denotational semantics underlying Hoare's programs which can justify his method. By avoiding the use of go-to statements, each syntactic entity in a program - an expression, a statement, a list of statements, a block - can be given a semantic specification (a binary relation between memory contents) solely in terms of the semantic relations of its immediate constituents. For example, if R_S denotes the relation associated with program fragment S , and M denotes the domain of memory-contents, we can define the semantics of the following constructs:

(i) if $S = S_1;S_2$ then $R_S = R_{S_1} \circ R_{S_2}$

(ii) if $S =$ while Q do S_1

then $R_S =$ repeat$(Q,S_1) \cap (M \times \sim Q)$ where repeat (Q,S_1) is the reflexive transitive closure of $R_{S_1} \cap (Q \times M)$

In this way we can specify the semantics of any program P as a set of binary relations on M , and so the definition agrees with the scheme of denotational definition described previously. However, in Hoare's method, he does not associate with each relation R_S a single assertion which is to extend it, but rather *two* assertions, $T_S, U_S \subseteq M$ say, which are to have the property that $[T_S]R_S \subseteq U_S$ ($T_S\{S\}U_S$ in Hoare's notation). Note that when S is the whole program P , U_P gives properties of the results of running P for inputs satisfying T_P .

To be a sound assertion method there must be some verification condition on the attached assertions which ensure the above property (which again we will call *validity*). For denotational definitions in general it seems difficult to formulate such a condition. However, for Hoare's go-to less programs, such conditions can be found relatively easily. Suppose we have, for each program fragment S , a sufficient condition for assertions T_S and U_S to be valid, in terms of the other assertions, *assuming they are valid*. In general such conditions would *not* constitute a consistent verification condition. However, if for each S , the condition for S is in terms of only assertions for *constituents* of S, then together they do form a consistent verification condition. This is because if assertions satisfy such conditions then their validity is assured by structural induction and the method is sound. For the method to be adequate, we have to be able to prove validity of any assertions T_P , U_P which *are* valid, by appropriate choice of other assertions. For this we require first that the conditions for each S also be <u>necessary</u> (i.e. if T_S, U_S are valid then there do exist other valid assertions that satisfy the condition for S) and secondly that no two conditions require the existence of valid assertions for the same program fragment (i.e. there is no conflict in choosing a single pair of assertions for each fragment). The latter condition is ensured by expressing the condition for S in terms of valid assertions for <u>immediate</u> constituents of S .

This will be clearer if we consider the two constructs given earlier.

Conditions for these constructs result from the following two identities:

(i) If $S = S_1;S_2$ then for all $A,B \subseteq M$

$$[A]R_S \subseteq B \text{ if and only if } (\exists C \subseteq M)\, [A]R_{S_1} \subseteq C \;\&\; [C]R_{S_2} \subseteq B \,.$$

(<u>Proof</u>: Note that $[A]R_S \subseteq B \iff [A]R_{S_1} \circ R_{S_2} \subseteq B \iff [[A]R_{S_1}]R_{S_2} \subseteq B$.

<u>'if' part</u>: $[A]R_{S_1} \subseteq C \;\&\; [C]R_{S_2} \subseteq B \Rightarrow [[A]R_{S_1}]R_{S_2} \subseteq B$

$$\Rightarrow [A]R_S \subseteq B$$

<u>'only if' part</u>: Let $C = [A]R_{S_1}$. Then $[A]R_{S_1} \subseteq C$ and

$$[A]R_S \subseteq B \Rightarrow [C]R_{S_2} \subseteq B \,.)$$

(ii) If S = <u>while</u> Q <u>do</u> S_1 then for all $A,B \subseteq M$

$$[A]R_S \subseteq B \text{ if and only if } \exists C \subseteq M,\ A \subseteq C,\ C \cap \sim Q \subseteq B \text{ and } [C \cap Q]R_{S_1} \subseteq C$$

(Proof: Note that $[A]R_S \subseteq B \iff [A](\underline{\text{repeat}}\ (Q, S_1) \cap (M \times \sim Q)) \subseteq B$

$\iff [A]\underline{\text{repeat}}\ (Q, S_1) \cap \sim Q \subseteq B$.

'if' part:

$[C \cap Q]R_{S_1} \subseteq C \implies [C](R_{S_1} \cap (Q \times M)) \subseteq C$

$\implies [C](R_{S_1} \cap (Q \times M))^{\textcircled{n}} \subseteq C$ for all $n \geq 0$.

$\implies [C]\ \underline{\text{repeat}}\ (Q, S_1) \subseteq C$.

Since $A \subseteq C$ and $C \cap \sim Q \subseteq B$,

$[A]\ \underline{\text{repeat}}\ (Q, S_1) \cap \sim Q \subseteq C \cap \sim Q \subseteq B$

$\therefore\ [A]R_S \subseteq B$.

'only if' part:

Let $C = [A]\underline{\text{repeat}}\ (Q, S_1)$. Then $C \cap \sim Q \subseteq B$ and, since repeat is reflexive, $A \subseteq C$.

$[C \cap Q]R_{S_1} = [C](R_{S_1} \cap (Q \times M))$

$= [A](\underline{\text{repeat}}\ (Q, S_1) \circ (R_{S_1} \cap (Q \times M))$

$\subseteq C$ by definition of $\underline{\text{repeat}}\ (Q, S_1)$.)

The 'if-parts' of such identities can be translated into conditions on the attached assertions T_{S_i}, U_{S_i} , which together would form a verification condition for the program in question. For example the first identity would give the condition

(i) $T_S = T_{S_1}\ \&\ U_S = U_{S_2}\ \&\ U_{S_1} = T_{S_2}$.

and the second would give the condition

(ii) $T_S \subseteq U_{S_1}$ & $U_{S_1} \cap \sim Q \subseteq U_S$ & $T_{S_1} = U_{S_1} \cap Q$.

The 'only if parts' of the identities ensure that the method is adequate. If T_S and U_S <u>are</u> valid then they can be proved valid by choosing assertions T_{S_1}, T_{S_2}, U_{S_1}, U_{S_2} etc. as in the proofs of the identities. No two conditions, for S, S' say will require choosing assertions for the same fragment S", since S" can be an immediate constituent of at most one of S and S' and the conditions are only in terms of immediate constituents. Thus a single choice of assertions for each fragment will satisfy all the condition simultaneously, i.e. **will satisfy the verification condition.**

We have noted that the verification conditions obtained must be consistent because a proof by structural induction, using the identities, could prove that assertions satisfying the conditions are valid. This being the case, rather than give the verification condition, it is sufficient to give the <u>identities</u>, as rules for <u>proving</u> validity. This is the approach of Hoare. It results in an elegant formal system for proving assertions valid.

Note however that in general it is not possible to simply replace verification conditions by inference rules in this way; it only works here because of the 'nested' structure of the semantic relations. This is therefore a powerful argument in favour of such 'structured' programming languages. Note also that the rules only need use the 'if' parts of the identities; however, the 'only if' parts are still necessary, for the the method to be adequate.

4: COMMENTS

When applying the assertion method to flowchart programs it is usual to attach only one assertion per loop. That is, we choose a 'cut-set' of points in the program at which to attach assertions. In keeping with the point of view of this paper, this corresponds to increasing the size of the basic semantic entities in our program from statements to loop-free pieces of flowchart. Such modifications of a language definition are not always possible. For example, if we consider 'parallel flowcharts' [2], increasing the size of the basic entities may eliminate possible computations by decreasing interactions between parallel processes. In this case the 'one assertion per loop' rule fails.

The correspondence between the assertion-method and the underlying semantics of the programming language implies that questions about the method, such as 'how do you handle overflow?' should really be asked about the underlying semantic definition. If the definition says that '+' for example is not truly addition of integers, but addition over some domain of 'finite precision integers', then this is the operation that must be used in proofs of assertions. Such proofs could well use theorems of numerical analysis. On the other hand if we use true integers in proofs then essentially we are considering an idealised language, without overflow.

5: REFERENCES

[1] Ashcroft E.A. 'Functional Programs as Axiomatic Theories', Centre for Computing and Automation, Report No. 9, Imperial College, London (1969).

[2] Ashcroft E.A. 'Proving Assertions about Parallel Programs', Research report CS-73-01, Dept. of Applied Analysis & Computer Science, University of Waterloo, Waterloo, Ontario (1973).

[3] Ashcroft E.A. 'A Situational Definition of a Large Subset of Algol-60', Research report CS-73-18, Dept. of Applied Analysis & Computer Science, University of Waterloo, Waterloo, Ontario (1973).

[4] Culik K. 'A Model for Formal Definition of Programming Languages', Research report CSRR 2065. Dept. of Applied Analysis & Computer Science, University of Waterloo, Waterloo, Ontario (1972).

[5] Floyd R.W. 'Assigning Meanings to Programs', Proc. Am. Math. Soc. Symposia in Appl. Math. 19 (1967).

[6] Hitchcock P. and Park D. 'Induction Rules and Termination Proofs'. Proceedings of IRIA Conf. on Theory of Automata and Prog. Languages (1972).

[7] Hoare C.A.R. 'An Axiomatic Approach to Computer Programming', Comm. ACM 12, No. 10 (1969).

[8] Hoare C.A.R. 'Procedures and Parameters: an Axiomatic Approach'. In Symposium on Semantics of Algorithmic Languages, lecture notes Mathematics, Vol. 188, E. Engeler (Ed.) Springer Verlag (1971).

[9] Lucas P. and Walk K. 'On the Formal Description of PL/I', Annual Review in Automatic Programming 6 (1969).

[10] Manna Z. 'Properties of Programs and First-Order Predicate Calculus', J. ACM 16, No. 2 (1969).

[11] Manna Z. 'The Correctness of Programs', J. Comput. Syst. Sci. 3, No. 2 (1969).

[12] Manna Z. and Vuillemin J. 'Fixpoint Approach to the Theory of Computation', Comm. ACM 15, No. 7 (1972).

[13] Manna Z. and Pnueli A. 'Formalization of Properties of Functional Programs', J. ACM 17, No. 3 (1970).

[14] McCarthy J. 'Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I', Comm. ACM 3, No. 4 (1960).

[15] McCarthy J. 'Lisp 1.5 Programmer's Manual', Computation Center and Research Laboratory of Electronics, M.I.T. (1962).

[16] Park D. 'Fixpoint Induction and proofs of Program Properties', Machine Intelligence 5. Edinburgh University Press (1969).