stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 144/80 SEPTEMBER
(DEPARTMENT OF COMPUTER SCIENCE)

R.J.R. BACK

CHECKING WHETHER PROGRAMS ARE CORRECT OR INCORRECT

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

1980 Mathematics subject classification: 68B05, 68B10

ACM-Computing Reviews-categoriey: 5.24

Checking whether programs are correct or incorrect.

by

R.J.R. Back

ABSTRACT

A program is considered as consisting of two parts, an algorithm part and a specification part, where the latter records the programmer's intentions as to how the algorithm should work. The question of what is a proper notion of semantic correctness for such programs is discussed, the emphasis being on finding a criterion which supports the construction and maintenance of correct programs. For a suitable notion of semantic correctness, a system to check it for programs is described. This system will either declare a program to be semantically correct, or it will show it to be incorrect by exhibiting a semantic error.

KEY WORDS & PHRASES: *program verification, partial correctness, total correctness, clean termination, run-time errors, loop invariants, multiple exits, transition diagrams, program construction, symbolic execution, testing, debugging, variable initialisation*

1. INTRODUCTION

In talking about program correctness one usually distinguishes between *syntactic* and *semantic* correctness. A program is *syntactically correct* if it is written according to the grammatical rules of the programming language used. If it also works as the programmer intended it to work, then the program is *semantically correct*. Syntactic correctness is in most cases checked mechanically by the compiler. Checking semantic correctness is, however, more difficult. A number of different techniques are used for this purpose, ranging from program testing to formal verification of program correctness.

One of the problems with checking semantic correctness is that the methods used are not strong enough to *decide* whether a program is semantically correct or not. Thus program testing can be used to show that a program is incorrect, by exhibiting an input for the program which does not produce the correct results. However, the fact that no errors are found by testing does not allow us to infer that the program is correct. Further testing might reveal an error not previously detected. The converse is true of proving program correctness. The fact that we are not able to prove a program correct does not justify the conclusion that the program is incorrect. The program could still be correct, but the proof of its correctness can be difficult to find.

One way to solve this problem is to strenghten the traditional methods so that it becomes possible to decide whether a program is correct or not. GOODENOUGH & GERHART [8] discuss how to strengthen the method of program testing so that the correctness of a program can be inferred from the fact that no errors are detected by testing. KATZ & MANNA [11] and BRAND [3] again describe how to extend verification techniques to proving that a program is incorrect. An alternative approach is to combine program testing and program proving, e.g. by first testing the program, and if no errors are found try to prove the program correct. This latter approach still leaves the possibility open that testing fails to produce an error while the correctness proof does not succeed either; in this case we still do not know whether the program is correct or not.

The situation with respect to semantic correctness should be compared with the way in which syntactic correctness of programs is checked. The checking is done by a compiler which analyzes the program and decides whether it is syntactically correct or not. Moreover, if the program is found to be incorrect, the compiler will indicate the place and nature of the syntactic error(s) responsible for the incorrectness. Something similar would clearly be desirable for checking semantic correctness of programs: a system which analyses a program and decides whether it is semantically correct or not, and in case of incorrectness indicates the semantic errors responsible for the incorrectness. Such a system would at the same time be a program verifier (proving semantic correctness), a program tester (proving semantic incorrectness) and a program debugger (locating the semantic error).

Obviously the program code alone does not contain enough information to enable one to determine whether a program is semantically correct or not, as this also depends on the intentions of the programmer. Checking semantic correctness does, however, become possible if these intentions are recorded as part of the program test. The system can, by inspecting the program text, then try to check whether the actual behaviour of the program is consistent with its intended behaviour. In this way semantic correctness is turned into an inherent property of the program, which only depends on the semantics of the programming language used. The situation is then the same as for syntactic correctness, which only depends on the syntax of the programming language used.

We will here pursue this approach to semantic correctness. We are aiming at a notion of semantic correctness which guarantees that the program works as intended by the programmer. This not only implies that the program must produce the correct results upon termination, but also that the program is guaranteed to terminate, and that the termination is *clean* (SITES [13]), i.e. the execution may not fail because of a run-time error. We will show how semantic correctness can be checked in a manner similar to the way in which a compiler checks syntactic correctness, and that the checking will decide whether the program is correct or not.

2. INVARIANT BASED PROGRAMS

An obvious candidate for semantic correctness is the *partial correctness* of programs. In this case the intentions of the programmer are expressed by the pre- and postconditions he provides for the program. Checking semantic correctness would thus amount to checking whether the program is partially correct with respect to these conditions.

The usual technique for proving partial correctness of iterative programs is to attach some suitably chosen *invariant* (*intermediate assertion*) to each loop in the program (FLOYD [6]). Using these invariants and the pre- and postconditions, a number of *verification conditions* are computed. If these verification conditions all hold, then the program will be partially correct.

This technique allows us to prove that a program is partially correct, but it does not allow us to *decide* whether a program is partially correct or not. To see this, consider the situation when some verification condition is found not to hold. From this we may infer that the program either is not partially correct, or that it is partially correct, but that the invariants were wrongly chosen. Thus the question whether it is partially correct or not is left open. Incorrectness can only be inferred from the fact that no choice of program invariants will make all verification conditions hold, a fact which is much more difficult to prove.

This argument should be sufficient to indicate that partial correctness does not lend itself easily to correctness checking along the lines desired. On the other hand, we may ask whether the programmer really wants this notion of semantic correctness. Consider again the situation in which the verification conditions do not hold. If the program in fact is partially correct then the invariants supplied by the programmer are wrongly chosen. But the programmer cannot know that the program is partially correct (this is what he is supposed to find out). The only thing he sees is that the program does not work in the way in which he thought it would work, where his ideas of how the program should work are expressed by the invariants he has provided for the program. The programmer, especially if he has constructed both program and invariants himself, is now as likely to suspect the program as the invariants of being wrong. He will feel free to change either one (or both) in order to achieve consistency between program and invariants (i.e. in order to make all verification conditions hold).

Thus the programmer is really interested in *consistency* between program and invariants. We should therefore take semantic correctness to mean consistency of this kind. The fact that some verification condition does not hold is then interpreted as a semantic error, which can be located to a specific part of the program (the part from which the verification condition was computed). This interpretation of semantic correctness does in fact provide us with a method for deciding correctness: a program will be semantically correct if and only if all verification conditions hold. (Of course we do not necessarily get a decision method in the *recursion theoretic* sense, as the truth of the verification conditions might not be decidable in the underlying theory.)

The price to be paid for this is that the invariants now have to be considered part of the program text. In other words, not only has the programmer to record his intentions as to what should be the pre- and postconditions of the program, but he also has to state his intentions as to how the postcondition is to be achieved by the program, by describing the appropriate intermediate assertions for the program. We will refer to programs of this kind as *invariant based programs.* A simple programming language in which to describe programs of this kind will be defined below. This language is a slight modification of the multi-exit statements previously described in BACK [2].

3. A LANGUAGE FOR INVARIANT BASED PROGRAMS

A simple programming language in which to express invariant based programs can be defined as follows. First we define *simple (multi-exit) statements* S. These are of the form

$$S ::= L \mid x_1,\ldots,x_k := e_1,\ldots,e_k;\ S_1 \mid \underline{\text{if}}\ b_1 \rightarrow S_1 \square \ldots \square\ b_k \rightarrow S_k\ \underline{\text{fi}} \quad (k \geq 1),$$

where $S_1,\ldots,S_k$ are simple statements, $x_1,\ldots,x_k$ are (distinct) program variables, $e_1,\ldots,e_k$ are expressions, $b_1,\ldots,b_k$ are boolean expressions and L is a label.

A *compound* (*multi-exit*) statement C has the form

$$C ::= \underline{\text{begin}}\ S_0 \blacksquare\ L_1 : S_1 \ldots \blacksquare\ L_k : S_k\ \underline{\text{end}} \qquad (k \geq 0),$$

where $S_0, S_1, \ldots, S_k$ are simple statements and $L_1, \ldots, L_k$ are (distinct) labels. For $k = 0$, C stands for the simple statement S_0.

A *declaration* D is of the form

$$D ::= \underline{\text{var}}\ x : T \mid \underline{\text{label}}\ L : Q$$

where x is a variable, T is an assertion (the *data invariant*), L is a label and Q is an assertion (the *label invariant*). An *environment* E is a sequence of declarations, i.e. it is of the form

$$E ::= D_1; \ldots; D_k, \qquad (k \geq 0).$$

A *block* B is of the form

$$B ::= E;\ C,$$

i.e. it consists of an environment E (the *local* environment) and a compound statement C.

Finally, an (*invariant based*) *program* H is of the form

$$H ::= E\{P\}B$$

where E is an environment (the *global* environment), P is an assertion (the *precondition*) and B is a block.

The simple multi-exit ststements are similar to Dijkstra's multiple assigned statements and guarded conditional statements [4]. The difference, as compared to Dijkstra's guarded commands, is that the syntax forces each execution of a simple statement to end in an explicit label L, signalling a jump to that label (i.e. L can be understood as *goto* L). Thus each simple statement has a single entry point but may have multiple exit points.

a compound multi-exit statement

$$\underline{\text{begin}}\ S_0 \blacksquare L_1 : S_1 \ldots \blacksquare L_k : S_k\ \underline{\text{end}}$$

corresponds to an ordinary Pascal block, with the symbol '$\blacksquare$' replacing ';'. The execution of this statement starts with S_0. If S_0 ends in one of the labels L_i, $1 \leq i \leq k$, then execution continues with the corresponding statement S_i, and so on. If S_0 ends in a label different from $L_1,\ldots,L_k$, the compound statement is exited (by a jump to that label). The order in which the labelled statements L_i: S_i are given in the compound statement does not influence the computation, as an execution of one labelled statement never can fall through to the next statement.

The programmer describes his intention by the declarations. The declaration <u>var</u> x : T states that any value assigned to x during the computation must satisfy the assertion T. Thus for instance

$$\underline{\text{var}}\ x : \text{integer}(x) \wedge 0 \leq x \leq 100$$

restricts x to vary in the set $\{0,1,\ldots,100\}$. This same restriction is expressed by the Pascal declaration (WIRTH [14]) <u>var</u> x : 0..100. However, data invariants are in general more powerful, in that any restriction on the values of the program variables is allowed. (The syntax of assertions will not be fixed here, but essentially we think of them as first-order formulas.) The declaration <u>label</u> L : Q states that we may assume Q to be true whenever execution of the program has reached label L. The syntax of the programming language is such that a loop can only be constructed using backward jumps to labels in a compound statement. It is thus illegal to program a loop without also giving the necessary intermediate assertion.

The program E{P}B contains all the information about the intended behaviour that is needed to check the semantic correction of it. The global environment E declares all global variables used in B. It also declares every possible exit L from the block B, together with the exit condition Q which must hold when exit L is taken (this is given in the form of a label declaration <u>label</u> L : Q in E). The precondition P states the condition on the global variables which may be assumed to hold initially.

4. CONSTRUCTING INVARIANT BASED PROGRAMS

In order to make the programming language above truly usable, we need to show how to construct invariant based programs in the first place. We will describe a technique in which one starts from the program invariants, giving a more or less formal description of these, and then tries to construct a program which respect these invariants. This approach reverses the usual order of program construction, in which the program is built first, and then one tries to discover the proper invariants. The approach outlined here has in various forms been considered by HOARE [10], DIJKSTRA [4], REYNOLDS [12] and VAN EMDEN [5]. The use of this program construction technique together with multi-exit statements has previously been described in BACK [1].

We will describe the programming technique and the use of multi-exit statements with the following simple lexical analysis problem: Let a line be a sequence of characters composed of letters and blanks only. A word is a sequence of letters only. The parse of a line is the sequence of words, in order, contained in the line. The words in the line are delimited by blanks or the end of the line. Our task is to construct a program for obtaining the parse of a line, given the line.

We start by fixing the global environment of the program. The input is given as the global variable ℓ, declared as

$\underline{\text{var}}\ \ell\text{: charseq}(\ell)$.

Here charseq(ℓ) is true iff ℓ is some sequence of characters. The output is given as the global variable p, declared as

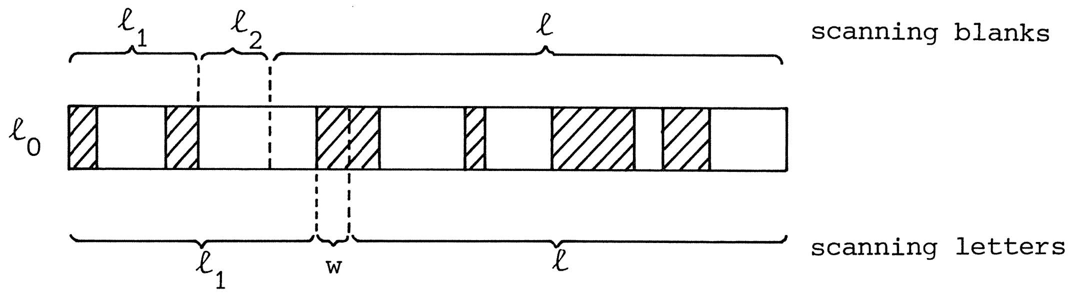
$\underline{\text{var}}\ p : \text{wordseq}(p)$,

where wordseq(p) is true iff p is a sequence of words.

The precondition will be that $\ell = \ell_0$, where ℓ_0 is the given sequence of letters and blanks. The purpose of the program is to compute the right value for the variable p, i.e. it must establish the situation *label* parse computed: p = parse of ℓ_0.

Here "parse computed" is a global label, to which the control is transferred when the computation is ready.

We will construct the obvious algorithm for solving this problem, i.e. we are going to scan the input line from left to right, and accumulate the words met in the variable p. There are two basic situations which repeatedly occur during the scan: either we are scanning blanks or we are scanning letters. These situations are illustrated in the following picture:



The shaded regions in the picture represent strings of letters, while the white regions represent strings of blanks. Consider first the situation in which we are scanning blanks. Part of the original line ℓ_0 has been scanned, and we can consider ℓ_0 as being built up of three consecutive strings, ℓ_1, ℓ_2 and the current line, i.e. $\ell_0 = \ell_1 \cdot \ell_2 \cdot \ell$ (the dot denotes concatenation). The words in ℓ_1 are already accumulated in the parse, i.e. p = parse of ℓ_1. The string ℓ_2 gives the blanks already scanned, i.e. ℓ_2 contains only blanks. Moreover, ℓ_2 must contain at least one blank, i.e. $\ell_2 \neq$ <>, otherwise we would not know that we are scanning blanks. This gives us the following invariant:

scanning blanks: $\ell_0 = \ell_1 \cdot \ell_2 \cdot \ell$, p = parse of ℓ_1, ℓ_2 contains only blanks and $\ell_2 \neq$ <>, for some strings ℓ_1 and ℓ_2.

For the other invariant we need an auxiliary variable w, declared as

<u>var</u> w: word(w),

in which we accumulate the word being scanned. A similar reasoning as the one above gives us the following invariant:

scanning letters: $\ell_0 = \ell_1 \cdot w \cdot \ell$, p = parse of ℓ_1, $w \neq$ <>, ℓ_1 = <> or last (ℓ_1) = ' ', for some string ℓ_1.

Here the condition ℓ_1 = <> or last (ℓ_1) = ' ' expresses the fact that w contains all the initial letters of the word being scanned.

From the initial situation ($\ell = \ell_0$) we reach one of these two invariants or the exit by the following simple multi-exit statement:

```
p := <>;
if $\ell$ = <> → parse computed
□ $\ell \neq$ <> → c, $\ell$ := first($\ell$),rest($\ell$);
            if c = ' ' → scanning blanks
            □ c ≠ ' ' → w := <c>; scanning letters
            fi
fi
```

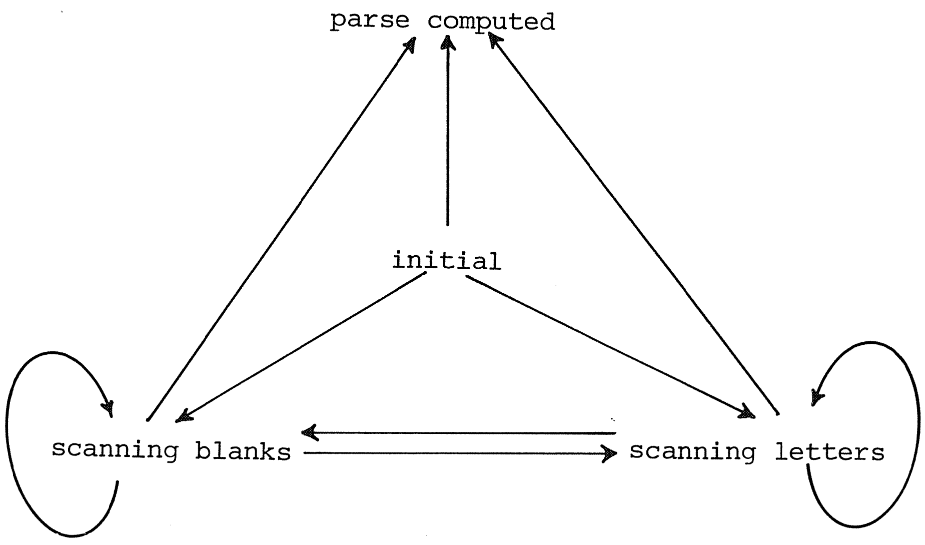
In a similar way we show how to proceed from the two invariants, scanning blanks and scanning letters. The algorithm for computing the parse is given by the following block:

```
var w: word(w);
var c: char(c);
label scanning blanks: $\exists \ell_1,\ell_2$ (charseq($\ell_1$), charseq($\ell_2$),$\ell_0 = \ell_1 \cdot \ell_2 \cdot \ell$,
                       p = parse of $\ell_2$, $\ell_2$ contains only blanks and
                       $\ell_2 \neq$ <>);
label scanning letters:$\exists \ell_1$(charseq($\ell_1$),$\ell_0 = \ell_1 \cdot w \cdot \ell$, p = parse of $\ell_1$,
                                  w ≠ <>, $\ell_1$ = <> or last($\ell_1$) = ' ');
begin p := <>;
      if $\ell$ = <> → parse computed
      □ $\ell \neq$ <> → c,$\ell$ := first($\ell$), rest($\ell$);
                  if c = ' ' → scanning blanks
                  □ c ≠ ' ' → w := <c> ;
                                  scanning letters fi fi
```

```
▮ scanning blanks:
      if ℓ = <> → parse computed
      □  ℓ ≠ <> → c,ℓ := first(ℓ), rest(ℓ);
                  if c = ' ' → scanning blanks
                  □  c ≠ ' ' → w := <c>;
                               scanning letters fi fi
▮ scanning letters:
      if ℓ = <> → p := p•<w>;
                  parse computed
      □  ℓ ≠ <> → c,ℓ := first(ℓ), rest(ℓ);
                  if c = ' ' → p := p•<w>;
                               scanning blanks
                  □  c ≠ ' ' → w := w•<c>;
                               scanning letters fi fi
  end
```

5. TRANSITION DIAGRAMS

An alternative way of describing invariant based programs is provided by *transition diagrams* (REYNOLDS [12], VAN EMDEN [5]). A transition diagram is a finite graph, where invariants are associated with the nodes of the graph and state transformations with the arcs of the graph. The program constructed in the previous section e.g. corresponds to the following transition diagram:



Here initial is a node associated with the initial situation ($\ell = \ell_0$). The state transformation associated with the transition from "scanning blanks" to "scanning letters" is e.g.

```
ℓ ≠ <>;
c,ℓ := first(ℓ), rest(ℓ);
c ≠ ' ';
w := <c>;
```

The guards in the conditional statements of the program thus correspond to partially defined identity transformations: $\ell \neq$ <> is e.g. a state transformation which is defined only if the condition $\ell \neq$ <> holds, and which does not change the values of any program variables.

The multi-exit statements provide a linear notation for transition diagrams. Such a linear notation is clearly to be preferred when the state transformations and/or the invariants become more complex and lengthy. The compound statement

$$\underline{\text{begin}}\ S_0 \blacksquare L_1 : S_1 \ldots \blacksquare L_k : S_k\ \underline{\text{end}}$$

bundles all transitions from label L_i together to form the single statement S_i ($L_0 \sim$ the initial state). The statement S_i is constructed by a carefully progressing *case-analysis* of the different possible situations which can occur when the invariant associated with L_i is known to hold initially. The syntax of simple statements is designed to make it possible to treat each case separately from the others, thus making the program construction task easier and the resulting program easier to understand and to modify.

6. SEMANTIC CORRECTNESS OF INVARIANT BASED PROGRAMS

We now turn to the question of defining a suitable notion of semantic correctness for invariant based programs. To motivate the definition to be given, we first look a little bit closer at the technique for constructing invariant based programs exemplified in the previous section.

The construction of a program E'{P}E:C starts by describing the global environment E' of the program. Let E' be the environment

$$\underline{\text{var}}\ y_1{:}U_1;\ldots;\underline{\text{var}}\ y_r{:}U_r;$$
$$\underline{\text{label}}\ K_1{:}R_1;\ldots;\underline{\text{label}}\ K_n{:}R_n.$$

Here $y_1,\ldots,y_r$ $(r\geq 1)$ are the global variables which the program will use. The labels $K_1,\ldots,K_n$ $(n\geq 1)$ are the possible exits of the program, while $R_1,\ldots,R_n$ are the postconditions associated with these exits. R_i is thus an assertion about the values of $y_1,\ldots,y_r$, which should be true when execution of the program terminates at label K_i. In addition to this we also have to state the precondition P of the program. This describes what we may assume to be initially of the values of the program variables $y_1,\ldots,y_r$.

The next step consists in setting up the local environment E for the program. This means that we have to determine the local variables which the program is going to manipulate, and that we also have to describe a number of intermediate stages of an algorithmic process aimed at establishing the postconditions of E'. Thus E will be of the form

$$\underline{\text{var}}\ x_1{:}T_1;\ldots;\underline{\text{var}}\ x_k{:}T_k;$$
$$\underline{\text{label}}\ L_1{:}Q_1;\ldots;\underline{\text{label}}\ L_m{:}Q_m{:}$$

where $x_1,\ldots,x_k$ are the local variables and $L_1,\ldots,L_m$ the local labels. The variable x_i may only be assigned values that satisfy condition T_i. The label L_i provides a name for an intermediate stage in the computation, while the associated invariant Q_i says what we may assume to be true of the values of the variables $x_1,\ldots,x_k$, $y_1,\ldots,y_r$ when this stage has been reached.

Finally we have to show how the computation can proceed from one intermediate stage to other stages (intermediate or final). This we do by providing for each intermediate stage L_i a single statement S_i which describes how the computation continues from this stage. Initialisation is provided by a statement S_0 which shows how to reach the intermediate stages (or directly a final stage) from a situation in which only the precondition P is known to hold. This gives us the compound statement C, where C is

$$\underline{\text{begin}}\ S_0 \blacksquare L_1 : S_1 \ldots \blacksquare L_m : S_m\ \underline{\text{end}}.$$

This completes the program construction task.

We may now ask whether the program constructed in this way is semantically correct. More precisely, is the initialisation S_0 correct and has each intermediate stage L_i been given a correct continuation S_i. A continuation S_i will be considered correct, if executing S_i for any initial state in which the (values of) the program variables $x_1, \ldots, x_k, y_1, \ldots, y_r$ satisfy the condition Q_i associated with the intermediate stage L_i (condition P in case of S_0), one of the exit labels of S_i is reached. Execution of S_i will not reach one of the exit labels if it fails because of a semantic error. A semantic error occurs at an assignment statement if an attempt is made to assign to a program variable a value which does not satisfy the data invariant associated with the variable. At a conditional statement a semantic error occurs if one of the guards is undefined or if none of the guards is true. Finally, a semantic error occurs at an exit label if the assertion associated with the label is not satisfied by the values of the program variables.

Thus, to summarise, we have the following definition. The program E'{P}E;C will be *semantically correct* if each simple statement S_i in C is correct, $i = 0,1,\ldots,m$. The simple statement S_i is *correct*, if for any initial state satifying condition Q_i $(Q_0 \equiv P)$, execution is guaranteed to reach an exit label of S_i, where Q_i is the condition associated with label L_i in E, $i = 1,\ldots,m$.

A program will essentially be semantically correct if and only if the verification conditions of the program are all satisfied. Thus semantic correctness implies partial correctness. It also implies that no run-time errors will occur. It does not, however, guarantee termination of the program. We will later show how to extend the notion of semantic correctness so that also termination of the program will be certain.

It might seem that a weaker notion of semantic correctness actually would be more appropriate: In the definition above, S_i would not be required to work correctly for any initial state satisfying Q_i, but only for those initial states which actually can be reached by some computation of C starting from an initial state satisfying precondition P. This is still stronger than partial correctness, because it implies that whenever a label is reached by the execution the assertion associated with the label will be true for the present state of the computation. For partial correctness it is only necessary that assertions associated with exit labels are satisfied when these labels are reached.

This weaker notion of semantic correctness will, however, be rejected for the following reason. Assume that the block B above is correct according to the weaker notion but not semantically correct according to the definition we have given above. This means that some simple statement S_i of C does not work correctly for some initial state s which satisfies condition Q_i, but that this state never can occur at label L_i during any execution of the block starting from a state satisfying condition P.

This means that the correctness of S_i depends on the not explicitly stated assumption that the state s never will be produced by the other simple statements in C (or by S_i itself). Consider now changing some other statement S_j in C, which has as one of its exits L_i. Changing S_j in a way which will produce state s at exit L_i but still results in the condition Q_i holding at this exit seems to be a perfectly valid change in the context of S_j. However, this change in one part of the program (S_j) will now produce an error in an other part of the program (S_i). This we consider to be highly undesirable and therefore choose to regard implicit assumptions of this kind as semantic errors.

Thus, by choosing to define semantic correctness in the way we did, we will get a correctness criterion which is robust with respect to changes made in the program. The interfaces between the different parts of the program are explicitly and completely stated in the form of label invariants. Therefore changes made in one part of the program will not affect the correctness of the other parts of the program, as long as the changes are consistent with the label invariants.

7. CHECKING SEMANTIC CORRECTNESS

The definition of semantic correctness given above shows that checking semantic correctness essentially amounts to checking that all the verification conditions are satisfied. One way of doing this is to compute all the verification conditions at once and ask the programmer (or maybe an automatic theorem prover) to prove them correct. The effect of this, however, is that the familiar program with which the programmer has been working now is changed into something very different, a set of theorems to be proved. Thus the programmer easily loses control of the verification process, finding it difficult to relate the correctness or incorrectness of the verification conditions to the semantic correctness of his program. We therefore prefer to have a technique for checking semantic correctness which works directly on the original program without transforming it into some different representation.

The technique to be presented here is based on *symbolic execution* (HANTLER & KING [9]) and is a *forward substitution technique* (GERHART [7]) for checking the correctness of verification conditions. This is preferred to a *backward substitution* technique, because it more closely resembles the usual method of hand simulating program execution, with which most programmers are familiar.

A program H is assumed to be of the form

$$E\{R \ \underline{\text{and}}\ x = t\}B,$$

where

E is the global environment,
R is an assertion,
x is a list of variables,
t is a list of terms and,
B is a statement (a block, compound statement or simple statement).

These will be subject to the following restrictions: E may not contain multiple declarations of the same identifier. No variable declared in E may occur free in R or occur in any term in t. The variables in the list x are all distinct, and every variable is declared in E. Finally, the lists x and t are of equal length.

The precondition should be read as

$$R \wedge x_1 = t_1 \wedge \ldots \wedge x_m = t_m,$$

where m is the length of the lists x and t. It asserts that the value of the variable x_i is initially given by the term t_i, $i = 1,\ldots,m$. This value of x_i is expressed in terms of some *symbolic constants*, i.e. auxiliary variables not declared in the environment E or in the program B. The assertion R states what we know about these symbolic constants. The use of symbolic constants makes it possible to relate the initial values of program variables to their values upon exit from the program. This is necessary in order to formulate the adequate postconditions for the program.

The example program built in section 3 (let us call it B) is of this form, i.e. it is

$$E\{\ell_0 \text{ contains only blanks and letters } \underline{\text{and}}\ \ell,p = \ell_0,p_0\}B,$$

where E is the environment

$\underline{\text{var}}\ \ell$: charseq($\ell$); $\underline{\text{var}}$ p: wordseq(ℓ);

$\underline{\text{label}}$ parse computed: p = parse of ℓ_0;

The system for checking semantic correctness will essentially be a proof system, i.e. a system for generating all semantically correct programs. This proof system will contain no axioms but a number of proof rules. A proof rule will be of the form

$$\frac{H_1,\ldots,H_n,F_1,\ldots,F_m}{H} \qquad (m,n \geq 0)$$

where $H_1,\ldots,H_n$ and H are programs and $F_1,\ldots,F_m$ are some other conditions (usually first order formulas). The proof rule says that H will be semantically correct if $H_1,\ldots,H_n$ all are semantically correct and if in addition the conditions $F_1,\ldots,F_m$ are satisfied.

We will depart from the standard way of writing a proof rule and write the proof rule above in the form

$$H \dashrightarrow H_1,\dots,H_n \ \underline{\text{when}}\ F_1,\dots,F_m$$

This notation emphasises an alternative way of reading a proof rule: to show H correct, we have to show that $H_1,\dots,H_n$ are all correct and that $F_1,\dots,F_m$ all hold (i.e. the question of whether H is correct is reduced to the question whether $H_1,\dots,H_n$ are all correct, assuming $F_1,\dots,F_m$ hold).

The following proof rules are given for checking semantic correctness of invariant based programs:

1. *variable declaration*

$$E\ \{R\ \underline{\text{and}}\ x = t\}\ \underline{\text{var}}\ y{:}T;\ B$$
$$\dashrightarrow E;\ \underline{\text{var}}\ y{:}\ T\ \{R \wedge T[y'/y]\ \underline{\text{and}}\ x,y = t,y'\}\ B$$

We assume that y is not declared in E before (this requirement could be expressed by a <u>when</u>-condition too). y' is some new symbolic constant by which the initial value of y is denoted. T[y'/y] denotes the formule we get by substituting y' for all free occurences of y in T. Thus the variable y is assumed to be initialized to some value y' satisfying the data invariant T.

2. *Label declaration*

$$E\ \{\ R\ \underline{\text{and}}\ x = t\}\ \underline{\text{label}}\ L{:}Q;\ B$$
$$\dashrightarrow E;\ \underline{\text{label}}\ L{:}\ Q\ \{\ R\ \underline{\text{and}}\ x = t\}\ B.$$

We assume that L is not declared in E before. A label declaration is simply moved into the environment, without any changes made to the precondition.

3. *Compound statements*

$$E\ \{\ R\ \underline{\text{and}}\ x = t\}\ \underline{\text{begin}}\ S_0\ \blacksquare\ L_1{:}S_1\dots\blacksquare\ L_m{:}S_m\ \underline{\text{end}}$$
$$\dashrightarrow E\ \{\ R\ \underline{\text{and}}\ x = t\}\ S_0\ ,$$
$$E\ \{\ Q_i[x'/x] \wedge T[x'/x]\ \underline{\text{and}}\ x = x'\}\ S_i,\ i = 1,\dots,m.$$

We assume that the labels $L_1,\dots,L_m$ all are declared in E, Q_i is the assertion associated with label L_i in E, $i = 1,\dots,m$, and $Q_i[x'/x]$ denotes the formula we get by simultaneously substituting the list of distinct fresh variables x' for the free occurrences of the variables x in Q_i. The assertion T stands for $T_1 \wedge\dots\wedge T_n$, where $x = x_1,\dots,x_n$ and T_i is the data invariant associated with variable x_i in E, $i = 1,\dots,n$. Correctness of a compound statement is thus checked according to the definition we gave, i.e. we check that the initialization and each simple statement in the compound statement is correct.

4. *Conditional statements*

$$\begin{aligned}
&E\ \{\ R\ \underline{\text{and}}\ x = t\}\ \underline{\text{if}}\ b_1 \rightarrow S_1 \square\dots\square\ b_k \rightarrow S_k\ \underline{\text{fi}}\\
&\text{--->}\ E\ \{\ R \wedge b_i[t/x]\ \underline{\text{and}}\ x = t\}\ S_i,\ i = 1,\dots,k\\
&\underline{\text{when}}\ R \wedge x = t \Rightarrow \text{bool}(b_i),\ i = 1,\dots,k,\\
&\qquad R \wedge x = t \Rightarrow b_1 \vee\dots\vee b_k.
\end{aligned}$$

Here $\text{bool}(b_i)$ expresses the requirement that the guard b_i should have a well-defined boolean value. We also require that one of the guards must be true, when executing the conditional statement.

5. *Assignment statements*

$$\begin{aligned}
&E\ \{\ R\ \underline{\text{and}}\ x = t\}\ y := e;\ S\\
&\text{--->}\ E\ \{\ R\ \underline{\text{and}}\ x = t'\ \}\ S\\
&\underline{\text{when}}\ R \wedge x = t \Rightarrow T[e/y].
\end{aligned}$$

We assume that each variable in y is declared in E. y must be a list $y_1,\dots,y_k$ of distinct variables of x, and e is a list of expressions assigned to variables in y. The list of terms t' is defined by

$$t_i' = \begin{cases} e_j[t/x], & \text{if } x_i = y_j \text{ for some } j = 1,\dots,k \\ t_i & \text{otherwise.} \end{cases}$$

The formula T stands for $T_1 \wedge\dots\wedge T_k$, where T_j is the data invariant associated with y_j in the environment E. Thus we require that each expression e_j assigned to y_j satisfies the data invariant associated with y_j.

6. *Labels*

E { R <u>and</u> x = t }

--->

<u>when</u> $R \wedge x = t \Rightarrow Q$.

We assume that L is declared in E. Q is the assertion associated with label L in the environment E. Thus, when reaching a label, we do not have to reduce the program any further, but can immediately check whether it is correct or not.

The proof rules given above provide us with a way of checking semantic correctness of a program which is very similar to the way in which the syntactic correctness of a program is checked by a recursive descent parser. The environment E corresponds to the symbol table of the parser. The <u>when</u>-conditions can be seen as the code generated by the parser (it compiles the program to a sequence of correctness checks). Finally, R and x are local variables of the parser which it needs in order to compute the correctness checks.

Alternatively we may regard the proof system as a formalization of the way in which the programmer checks the correctness of his program by symbolic execution. The correctness formula E { R <u>and</u> x = t } B will then indicate a place in a program (the position immediately preceeding B). E gives the declarations valid at this place in the program, B is the part of the program which still has to be checked for correctness, and $R \wedge x = t$ states what we know to be true at this place in the program. Thus {R <u>and</u> x = t} serves as a marker. The proof rules show under which conditions the marker may be moved forward in the program text, or removed from the program when a label has been reached. If a marker can not be moved forward or removed, because a <u>when</u> - condition is violated, then a semantic error has been detected and the marker indicates the place of this error.

The nature of the semantic error is determined by the specific condition which is violated. A program will be correct if and only if all markers introduced into the program by the proof rules eventually can be removed.

This proof system is thus seen to satisfy the basic requirements we stated for checking semantic correctness. It can be used to determine whether the program is semantically correct or not, and in case it is not correct it will indicate the place and nature of a semantic error responsible for the incorrectness. (The use of this proof system to check the correctness of the example program of section 4 is illustrated in the appendix).

8. TERMINATION

The notion of semantic correctness defined in section 4 did not require computation of a block to terminate. This allows one to give trivial solutions to program construction problems. Thus if we e.g. are given an external environment E' and a precondition P, and have constructed the local environment E with local labels $L_1,\dots,L_k$, then the compound statement

$$\underline{\text{begin}}\ S_0 \blacksquare L_1{:}L_1 \ \dots\ \blacksquare L_k{:}L_k\ \underline{\text{end}}$$

will be a semantically correct solution, provided the intialization S_0 is correct. However, this compound statement will in most cases not terminate, making the solution useless.

It therefore is necessary to include in the criteria of semantic correctness a guarantee that the program will terminate. The most important technique for showing termination is due to FLOYD [6] and is based on the use of well-founded sets. One tries to find a function on the program variables which takes its values in a well-founded set, such that each execution of a loop in the program will decrease the value of this function. As the value of the function only can be decreased a finite number of times, this implies that the program must terminate.

We are now faced with a situation similar to the one encountered earlier w.r.t. partial correctness: The proof method does not allow one to decide whether the program terminates or not. If the chosen termination function is decreased by each loop in the program, this will indeed prove that the program always terminates. If, however, there is some loop which does not decrease the termination function, we are not allowed to infer that the program does not always terminate. It is still possible that the program always terminates, but that the proof failed because the wrong termination function was chosen.

As before, we can now argue that the programmer, when faced with a situation where the chosen termination function is not decreased by some loop, is as likely to suspect the program to contain an error as he is to suspect the termination function to be wrongly chosen. If for instance the choice of termination function is made before the compound statement is constructed, then the transitions in the compound statement should be designed in such a way that no nondecreasing loop is ever introduced.
If such a loop is nevertheless introduced, then some transition must be wrong. The programmer will try to achieve consistency between program and termination function by changing the program and/or the termination function in such a way that each loop in the program will decrease the value of the termination function.

We will require that the programmer records his choice of termination function in the program text. For this purpose we add a new kind of declaration to our programming language. This declaration has the form

D::= $\underline{\text{decrease}}$ h,

where h is some integer valued expression (the *termination function*). At most one such declaration is allowed in a block.

We now have to extend the notion of semantic correctness to programs containing a declaration of a termination functions. Consider the program H, of the form

E'{P}E; $\underline{\text{decrease}}$ h;C.

Then H is *semantically correct* if E'{P}E;C is semantically correct according to the previous definition, and if in addition h is finitely decreasing in B = E;C.

The expression h is said to be *finitely decreasing* in B, if the following three conditions are satisfied:

(i) The expression h has a well-defined non-negative value in any program state satisfying some label invariant of B.

(ii) No transition in C from one internal label to another internal label can increase the value of h.

(iii) Every cycle in the transition diagram of C contains at least one transition which is guaranteed to decrease the value of h.

Obviously any semantically correct program with a termination function declaration will be totally correct with respect to the given precondition and environment. As explained above, the converse does not necessarily hold.

Similar objections can be raised against this way of defining semantic correctness with termination as was raised against our definition of semantic correctness in section 4. Thus one might argue that the requirements are too strong, and that a weaker requirement for termination would be more appropriate. More precisely, one would only require that h is decreased in cycles which actually can be traversed by some execution of the program.

One example of such a cycle is provided by the following block.

```
var x: integer(x);
label L: x ≥ 0;
begin x := 0; L
■ L: if x < 0 → L
       □ x ≥ 0 → L' fi
end
```

The corresponding transition diagram is

$$\begin{tikzcd}
\text{initial} \arrow[d, "x{:=}0"] \\
L \arrow[loop right, "x<0"] \arrow[d, "x\geq 0"] \\
L'
\end{tikzcd}$$

Here it is obvious that the branch guarded by the condition $x < 0$ can never be taken, and consequently the cycle in the transition diagram will never be traversed by an execution.

The question now is whether such branches should be allowed or not. If we regard them as errors, then it is not really important whether h is decreased by the cycle or not, because the program will be incorrect anyway.

Such branches can be detected by adding to each reduction of a formula $E\ \{\ R\ \underline{\text{and}}\ x = t\}\ B$ the check $\exists x'.R$, where x' are all the free variables occurring in R. This condition will be true iff the place indicated by the correctness formula actually can be reached from some initial state satisfying the assertion associated with the preceding label. In the example above, the correctness formula

$$E\ \{\ x' \geq 0 \wedge x' < 0\ \underline{\text{and}}\ x = x'\}\ L$$

would eventually be generated, and the fact that the cycle from L to L cannot be traversed would be detected by noticing that

$$\exists x'(x' \geq 0 \wedge x' < 0)$$

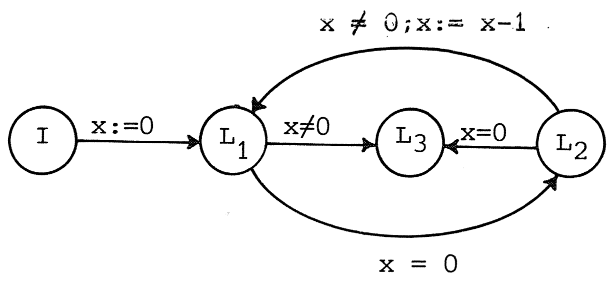
does *not* hold.

On the other hand, we may choose not to regard the situation above as an error. As the branch in question cannot be traversed, it is of course redundant, but it does not do any harm either. It is, however, conceivable that the branch later will be changed in some way which makes the cycle traversable. If we have not required h to be decreased by this cycle in the program, then the change might introduce a nonterminating loop into the program. Changing e.g. the guard $x < 0$ in the example program to $x \leq 0$ seems to be a perfectly legal change and the resulting program will still be semantically correct w.r.t. the label invariants. It will not, however, be guaranteed to terminate anymore.

In the example above the proof system was able to detect the fact that the cycle could not be traversed. This is not anymore the case in the following block

```
var x: integer(x); label L1: x ≥ 0; label L2: x ≥ 0;
begin x := 0; L1
▮ L1: if x = 0 → L2
          □ x ≠ 0 → L3 fi
▮ L2: if x = 0 → L3
          □ x ≠ 0 → x := x-1; L1 fi
end
```

The corresponding transition diagram is as follows:



Obviously L_2 is only reached when $x = 0$ holds, so that the branch from L_2 guarded by $x \neq 0$ cannot ever be taken, i.e. there is no traversable cycle in the program. Using the weaker requirement for termination, we would in this case not need any termination function at all. However, it is easy to introduce an infinite loop into this program by what appears to be a perfectly legal change in the program. We can do this by e.g. changing the simple statement associated with L_1 to

```
L1: if x = 0 → x := x+1; L2
    □ x ≠ 0 → L3           fi.
```

This change will respect all the label invariants, but the program will never terminate.

The weaker requirement with respect to termination is too sensitive to changes in the program and will be rejected for this reason:changes which appear to be correct in the local context in which they are made may nevertheless introduce nonterminating loops into the program. The criterion of semantic correctness for termination that we adopted is much more robust in this respect. Any change made to a program must respect both the invariants and the termination function. As long as a local change does not affect the cycle structure of the program, this is sufficient to guarantee that the modified program also is semantically correct.

9. CHECKING TERMINATION

Having extended the notion of semantic correctness to also cover termination, we now need to change the proof rules of section 5. We have to check whether the termination function of a block actually is finitely decreasing in the block. The requirement that the termination function is well-defined and non-negative for each label invariant is easy to check, as well as the requirement that the value of L is not increased by any transition. However, checking that each cycle of a block contains a transition which actually decreases the value of the termination function is a little bit more difficult.

We will deal with this last requirement as follows. Let C be a compound statement

$$C = \underline{\text{begin}}\ S_0 \blacksquare L_1 : S_1 \ldots \blacksquare L_k : S_k\ \underline{\text{end}}.$$

We will now require that the labels $L_1, \ldots, L_k$ are ordered in such a way that each <u>backward transition</u> is quaranteed to decrease the value of the termination function. A backward transition is a transition from a label L_i to a label L_j, where $j \le i$. As each cycle in C must contain at least one backward transition, this requirement is sufficient to guarantee that each cycle in C contains at least one transition which will decrease h.

On the other hand, this requirement could be too strong. To prove that this is not the case, we will show that each compound statement

$$C = \underline{\text{begin}}\ S_0 \blacksquare L_1 : S_1 \ldots \blacksquare L_k : S_k\ \underline{\text{end}}$$

in which the termination function h is finitely decreasing can be changed into an equivalent statement

$$C' = \underline{\text{begin}}\ S_0 \blacksquare L_{i_1} : S_{i_1} \ldots \blacksquare L_{i_k} : S_{i_k}\ \underline{\text{end}},$$

in which each backward jump decreases h, by simply permuting the order of the simple statements in C.

Let C be a compound statement in which h is decreasing. The labels in C' are ordered by the following procedure: Let A be a set of label identifiers, which we initialize to contain all global exit labels occurring in C. We choose L_{i_k} to be any label s.t. each transition from L_{i_k} either decreases the value of h, or goes to a label in A. We then add L_{i_k} to the set A. $L_{i_{k-1}}$ is then chosen in the same way (i.e. each transition from $L_{i_{k-1}}$ either decreases h or goes to a label in A) and added to A. This is continued until each label $L_1,\dots,L_k$ has been added to A. The compound statement C' determined by this ordering of labels in C will have the required property, i.e. each backward transition will decrease the value of h. (This is easily established by induction).

We also need to show that each label $L_1,\dots,L_k$ eventually will be included in the set A. Assume that this is not the case, i.e. that after a certain number of steps we are left with a nonempty subset B of labels in $\{L_1,\dots,L_k\}$, such that none of the labels in B can be added to A. Let L_1' be some label in B. Then there must be a nondecreasing transition from L_1' to a label in B, say L_2', otherwise L_1' could be included in A. The same holds for L_2', i.e. there must be a nondecreasing transition from L_2' to some L_3' in B, and so on. Thus there is an infinite sequence L_1', L_2', L_3', ... of labels in B, such that there is a nondecreasing transition from L_i' to L_{i+1}', for $i = 1,2,\dots$. But there can only be a finite number of labels in B. Consequently L_i' must be equal to L_j' for some $i < j$, i.e. C contains a cycle in which no transition decreases h. This contradicts the assumption about C, thus each label must eventually be included in A.

The proof rules required for checking termination can now be given. First, we need a proof rule for the declaration of a termination function. The proof rule is similar to the rule for label declaration, i.e. the declaration is simply added to the environment.

7. *Termination function*

```
E { R and x = t} decrease h; B
----> E; decrease h { R and x = t } B.
```

We also need to change the proof rule for compound statements, to check whether the termination function is finitely decreasing. We have the following proof rule:

3'. *Compound statement with termination*

$$E\ \{\ R\ \underline{\text{and}}\ x = t\}\ \underline{\text{begin}}\ S_0\ \blacksquare\ L_1\ :\ S_1\ \ldots\ \blacksquare\ L_k\ :\ S_k\ \underline{\text{end}}$$
$$\text{--->}\ E\ \{\ R\ \underline{\text{and}}\ x = t\}\ S_0,$$
$$E_i\ \{\ Q_i[x'/x] \wedge T[x'/x] \wedge h = h'\ \underline{\text{and}}\ x = x'\}\ S_i,\ \text{for } i = 1,\ldots,k,$$
$$\underline{\text{when}}\ Q_i \Rightarrow h \geq 0,\ \text{for } i = 1,\ldots,k.$$

Here Q_i is as before the assertion associated with the label L_i in E, h is the termination function in E, x' is a list of fresh variables and h' is some fresh variable. The environments E_i are defined as follows, for $i = 1,\ldots,k$. Let E' be the environment from which the declarations of the labels $L_1,\ldots,L_k$ have been deleted. Then

$$E_i = E';\ \underline{\text{label}}\ L_1 : Q_1 \wedge h < h';\ldots;\ \underline{\text{label}}\ L_i : Q_i \wedge h < h';$$
$$\underline{\text{label}}\ L_{i+1} : Q_{i+1} \wedge h \leq h';\ldots;\ \underline{\text{label}}\ L_k : Q_k \wedge h \leq h';$$

In this proof rule we thus check that each backward jump from an internal label does decrease the value of h, and that no forward jump to another internal label increases the value of h. No restrictions need to be put on the transitions associated with the initialization statement S_0 or on the transitions leading out of the block, to external labels.

This proof rule will report an error for compound statements in which h is finitely decreasing, but where the labels are in the wrong order. As shown above, it is a straightforward matter to convert such a compound statement to a correct one by simply permuting the ordering of the labels. We believe that the added clarity of the program structure together with the better control the programmer has over termination when following the restrictions set by this rule outweighs the inconveniencies caused by the restrictions.

The question of proving termination of state transition diagrams is discussed both by VAN EMDEN [5] and by REYNOLDS [12]. The former identifies the usual conditions needed for establishing termination, i.e. that the termination function must be finitely decreasing in the transition diagram. Reynolds goes one step further and proposes that the geometrical structure of the transition diagram is chosen so that each cycle is readily identified.

Reynolds uses the principle that a transition which increases the distance from the origin (the initial situation) also must increase the information content, i.e. the target node has more information than the source node. Transitions which do not increase information must decrease the value of the termination function. The net effect is the same as the one we get by the proof rule above: backward transitions must decrease the termination function. The two-dimensional structure of transition diagrams is here an advantage, making the cycle structure of the program easy to recognise. Finally, some other ways of handling termination are described in BACK [1] and BACK [2].

10. INITIALIZATION OF VARIABLES

The proof rule given for variable declarations assumes that a variable y declared by

$\underline{\text{var}}$ y: T,

is assigned some not further specified initial value satisfying the data invariant T. This is a simplifying but not very realistic assumption, so we will here discuss two alternative ways of handling variable declarations.

The first approach would be to assign to a variable some explicit initial value upon declaration. We could change the declaration of a variable to be of the form

$\underline{\text{var}}$ y := e: T,

where e would be an expression giving the initial value of y. This expression could be allowed to depend on values of global variables and variables declared before y.

In this case one would have to change the proof rule for variable declaration. The new proof rule would be as follows.

1'. *Variable declaration (with explicit initialization)*

E { R <u>and</u> x = t} <u>var</u> y : = e: T; B
---> E; <u>var</u> y: T { R <u>and</u> x,y = t,e[t/x]} B
<u>when</u> R $\wedge$ x = t $\Rightarrow$ T[e/y].

No other proof rule would have to be changed in this case.

Alternatively we could assume that a variable is only initialized when it is assigned a proper value by an assignment statement. In this case we would give the following proof rule for a declaration of a variable:

1". *Variable declaration (without initialization)*

E { R <u>and</u> x = t} <u>var</u> y: T; B
---> E; <u>var</u> y: T { R <u>and</u> x = t} B.

In this case the list x of variables in the precondition only mentions those variables which have been properly initialized. Thus a variable declaration does not add a new variable to this list, this is done only when the variable is first assigned a proper value.

This approach requires us also to change the proof rule for assignment statements. The new proof rule is

5". *Assignment statement (with initialization)*

E { R <u>and</u> x = t} y := e; S
---> E { R <u>and</u> x' = t' } S
<u>when</u> R $\wedge$ x = t $\Rightarrow$ T[e/y].

If y = x_i for some i, $1 \le i \le n$ (n is the length of list x) then x' = x and t' = $t_1, \ldots, t_{i-1}$, e[t/x], $t_{i+1}, \ldots, t_n$. Otherwise x' = $x_1, \ldots, x_n$, y and t' = $t_1, \ldots, t_n$, e[t/x].

Finally we also have to change the rule for compound statements. For each label in the compound statement we need to know exactly which variables may be assumed to be initialized. We could e.g. require that each variable y assumed to be initialized at a label is specified as such be adding the assertion T(y) to the corresponding invariant, where T is the data invariant associated with the variable y. This, however, will make the invariants lengthy and writing them tedious. It would therefore be nice to have some default convention, by which the variables assumed to have well defined values are implicitly stated in the invariant. A reasonable default convention is to assume that each variable occurring free in the invariant has a well-defined value, and thus must have been initialized before the corresponding label has been reached. A variable y which does not occur free in an invariant, but which we still wish to assume being initialized, can be stated as having a well defined value by adding the assertion T(y) to the invariant.

With this convention the proof rule for the coumpound statement is changed from the form given in section 5 to the following:

3". *Compound statement (with initialization)*

$$E \{ R \ \underline{\text{and}}\ x = t\} \ \underline{\text{begin}}\ S_0 \blacksquare L_1 : S_1 \ldots \blacksquare L_k : S_k \ \underline{\text{end}}$$

$$\dashrightarrow E \{ R \ \underline{\text{and}}\ x = t\}\ S_0$$

$$E \{ Q_i[z_0{}^i/z] \wedge T[z_0{}^i/z] \ \underline{\text{and}}\ z = z_0^i \}\ S_i, \quad i = 1,\ldots,k.$$

Here Q_i is the label invariant associated with label L_i in E, z^i is the list of variables occurring free in Q_i and z_0^i is a list of fresh constants.

This second approach makes it an error to refer in an expression to the value of a variable before this variable has been initialized to some proper value. Such an error will be caught by the modified proof system (containing the proof rules 1", 5" and 3"). The error will be caught in an assignment statement or a conditional statement by the impossibility of proving that the when-condition asserting that the expression (or the boolean expression) in question has a well-defined value.

In e.g. the rule for assignment statements the when-condition has the form

$$R \wedge x = t \Rightarrow T[e/y].$$

Assume that e contains a reference to a variable z which has not been properly initialized. This variable does therefore not occur in the list x. Neither can it occur in the assertion R, because R may not contain free occurrences of any variable declared in a program or in its global environment. Consequently, nothing is known about the value of z, and the fact that T[e/y] holds can therefore not be proved.

11. SUMMARY

This report has emphasized a view of programs which can be expressed by the quasi-definition

program = algorithm + specification.

The specification part records the intentions of the programmer as to how the algorithm should behave, while the actual behavior of the algorithm is determined by the semantics of the programming language used.

The advantage of this viewpoint is that semantic correctness becomes an intrinsic property of programs, only depending on the semantics of the programming language used (which also must provide a meaning for the specification part of the program). The situation is thus similar to syntactic correctness, which also is an intrinsic property of programs, only depending on the syntax of the programming language.

One of the main problems treated in this report centers around the question of exactly what notion of semantic correctness should be chosen. We have used as our main criterion in selecting a suitable notion that the notion should support the construction and maintenance of programs. This means that it should be easy for the programmer to convince himself of the correctness of the program he has designed, and also that it should be easy to check that a change in the program preserves the correctness.

The notion of semantic correctness chosen in this report is such that a semantically correct program will be guaranteed to terminate cleanly (i.e. termination is not caused by a run-time error), producing the desired results on termination. Thus semantic correctness implies total correctness. The converse, however, is not true, i.e. there are programs which are totally correct but which are not considered semantically correct. The kind of programs that fall into this category are programs which work correctly "for some mysterious reason". These programs do not work in the way the programmer intended them to work, but still manage to produce the right results. They are excluded both because they are difficult to maintain and because their correctness cannot be checked in a simple way.

Our notion of semantic correctness is intimately tied to the specific kind of programs we study, the *invariant based programs*. The algorithm part of such programs is expressed in a simple iterative language with unrestricted flow of control, while the specification part gives all the necessary data and label invariants, together with a termination function. The main task of this report has been to design a system by which the semantic correctness of invariant based programs can be checked. The system built will check semantic correctness in a way which is analoguous to the way in which a compiler checks the syntactic correctness of a program: The system analyses the program line by line, checking whether there is a semantic error in the program. The program will be semantically correct if and only if no semantic error is found by the system. In case an error is found, the system will indicate the place of the error in the program. The system will thus *decide* (relative to an oracle deciding the validity of assertions) whether a program is semantically correct or not. In doing this, it actually carries out the tasks usually performed by three separate systems: a program verifier (proving that a program is correct), a program tester (proving that a program is incorrect) and a program debugger (locating an error in the program).

APPENDIX

Below we give the example program of section 4 in full. The program checking system of section 7, together with the proof rules for termination in section 9 assigns to each place in the program block information about what is known about the values of the program variables, together with a condition to be checked to make sure that there is no semantic error at the place in question. Below we show the information and the checks for some selected places in the program. (The proof rule for termination of a block requires the local environment to be altered. The additions to the local environment are shown in square brackets, and corresponds to the situation when the transition from label "scanning letters" is being checked for correctness.)

```
var $\ell$: charseq ($\ell$);
var p: wordseq (p);
label parse computed: p = parse of $\ell_0$;
{$\ell_0$ contains only blanks and letters and $\ell$,p = $\ell_0$,$p_0$}
var w: word (w);
var c: char (c);
{$\ell_0$ contains only blanks and letters $\wedge$ word($w_0$) $\wedge$ char($c_0$) and
                                          $\ell$,p,w,c = $\ell_0$,$p_0$,$w_0$,$c_0$}
label scanning blanks: $\exists\ell_1,\ell_2$(charseq($\ell_1$) $\wedge$ charseq($\ell_2$) $\wedge$ $\ell_0 = \ell_1 \cdot \ell_2 \cdot \ell$ $\wedge$
                       p = parse of $\ell_1$ $\wedge$ $\ell_2$ contains only blanks $\wedge$ $\ell_2 \neq$ <> $\wedge$
                       [length($\ell$) < length($\ell'$)]);
label scanning letters: $\exists\ell_1$ (charseq($\ell_1$) $\wedge$ $\ell_0 = \ell_1 \cdot w \cdot \ell$ $\wedge$ p = parse of $\ell_1$ $\wedge$ w $\neq$ <>
                        $\wedge$ $\ell_1$ = <> or last($\ell_1$) = '' $\wedge$ [length[$\ell$] $\leq$ length[$\ell'$]);
decrease length ($\ell$);
```

```
begin p:= <>
      if $\ell$ = <> → parse computed
      □ $\ell \neq$ <> → {$\ell_0$ contains only blank and letters ∧ word($w_0$) ∧ char($c_0$) ∧
                        $\ell_0 \neq$ <> and $\ell$,p,w,c = $\ell_0$,<>,$w_0$,$c_0$; check that
                        char(first($\ell_0$)) ∧ charseq(rest($\ell_0$))}
                        c,$\ell$:= first($\ell$), rest($\ell$);
                        if c = '' → scanning blanks
                        □ c ≠ '' → w:= <c>;
                                    {$\ell_0$ contains only blanks and letters ∧
                                     word($w_0$) ∧ char($c_0$) ∧ $\ell_0 \neq$ <> ∧ first($\ell_0$) ≠ ''
                                     and $\ell$,p,w,c = rest($\ell_0$),<>,$w_0$,first($\ell_0$);
                                     check that 'scanning letters' holds}
                                     scanning letters fi fi
■scanning blanks:
      if $\ell$ = <> → parse computed
      □ $\ell \neq$ <> → c,$\ell$:= first($\ell$),rest($\ell$);
                   if c = '' → scanning blanks
                   □ c ≠ '' → w:= <c>;
                               scanning letters fi fi
■scanning letters:
      {'scanning letters'[$\ell$',p',w',c'/$\ell$,p,w,c] ∧ charseq($\ell$') ∧ wordseq(p')
       ∧ word(w') ∧ char(c') ∧ length($\ell$) = h' and $\ell$,p,w,c = $\ell$',p',w',c';
       check that length($\ell$) ≥ 0}
       if $\ell$ = <> → p:= p•<w>;
                        parse computed
       □ $\ell \neq$ <> → c,$\ell$:= first($\ell$),rest($\ell$);
                        if c = '' → p:= p•<w>;
                                    scanning blanks
                        □ c ≠ '' → w:= w•<c>;
                                    scanning letters fi fi
end
```

ACKNOWLEDGEMENT

I would like to thank J.W. de Bakker and H.B.M. Jonkers for their critical comments on an earlier draft of this paper, and A. de Bruin and J.V. Tucker for the stimulating discussions we had on the topics of this report.

REFERENCES

[1] BACK, R.J.R., *Program construction by situation analysis*, Computing Centre of University of Helsinki, Research Report 6, 1978.

[2] BACK, R.J.R., *Exception handling with multi-exit statements*, Programmiersprachen und Programmentwicklungen, Darmstadt 1980. Informtik-Fachbereich 25, Springer Verlag.

[3] BRAND, D., *Path calculus in program verification*, Journal of ACM, vol. 25, no. 4, October 1978, pp. 630-651.

[4] DIJKSTRA, E.W., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1976.

[5] VAN EMDEN, M.H., *Programming with verification conditions,* IEEE Transactions on Software Engineering, SE-5,2, 1979.

[6] FLOYD, R.W., *Assigning meanings to programs*, Proc. Amer. Math. Soc. Symp. in Applied Mathematics 19, 1967, pp. 19-31.

[7] GERHART, S.L., *Proof theory of partial correctness verification systems,* SIAM J. Computing, vol. 5, no. 3, September 1976, pp. 355-377.

[8] GOODENOUGH, J.B. & S.L. GERHART, *Towards a theory of test data selection*, Proc. Int. Conf. on Reliable Software, SIGPLAN Notices 10, June 1975, pp. 528-533.

[9] HANTLER, S.L. & J.C. KING, *An introduction to proving the correctness of programs*, Computing Surveys 8,3, 1976, pp. 331-353.

[10] HOARE, C.A.R., *Proof of a program: FIND*, Comm. ACM 14, January 1971, pp. 39-45.

[11] KATZ, S.M. & Z. MANNA, *Logical analysis of programs*, Comm. ACM 19, April 1976, pp. 185-206.

[12] REYNOLDS, J.C., *Programming with transition diagrams*, in: Gries, D. (ed.), Programming Methodology, Springer Verlag, Berlin, 1978.

[13] SITES, R.L., *Proving that computer programs terminate cleanly*. Ph. D. Thesis, Dept. of Comp. Science, Stanford U., STAN-CS-74-418, May 1974.

[14] WIRTH, N., *The programming language Pascal*, Acta Informatica 1, 1971, pp. 35-63.