Algorithms

10. Correctness of Programs

R K Shyamasundar

In the previous article, we studied notions such as algorithmic universality, computability, and incomputability. In this article, we study the notion of correctness of programs, illustrate a method of establishing the correctness of programs and discuss issues in the development of correct software.

R K Shyamasundar is a Professor of Computer Science at TIFR, Mumbai and has done extensive research in various foundation areas of computer science.

This article concludes the series.
Previous articles of this series were:
1. Introduction to algorithms, January 1996.
2. The while-construct, March 1996.
3. Procedures and recursion, June 1996.
4. Turtle graphics, September 1996.
5. Data types and their representation in memory, December 1996.
6. Algorithms for sorting and searching, March 1997.
7. Data structures: lists, queues, stacks and arrays, June 1997.
8. Algorithm design techniques, August 1997.
9. Universality and Incomputability, October 1997.

Introduction

The issue of establishing that computer programs *behave* as 'intended' has existed ever since the beginning of modern computers. In fact, A M Turing (See *Resonance*, Vol.2, No.7, 2–4; No.8, 50–59, 1997). the distinguished computer pioneer, describes a method of checking a large program in his early works. The importance of methods of establishing the correctness of programs has grown with the growth in complexity of programs or software. In general, 'correctness of a system' is the property which shows that the system *behaves* as *intended*. Thus, in the context of programming, this corresponds to showing that the program works as intended for the whole input domain. If the input domain is finite, then one can *test* the program for all the values. However, the input domain is usually/practically infinite. For example, if we have written a program to check whether a given number is prime, and we have tested it for values 2, 3, 5, 7 then we cannot conclude that the program works properly on the number 9 since we cannot use *interpolation* as in the classical systems to verify the design. Thus, *testing* can only show the *presence of errors* and not the *absence of errors*. Hence, it is very important to show that the program developed is indeed correct. Most of the time, we use 'correctness' in a relative manner. For instance, in the context of programs it is enough if we can establish that the *behaviour* of the program satisfies the

Our aim is to present the concepts assuming just the basic knowledge of the execution of a program and basic mathematical concepts.

specifications (intentions). In the sequel, we shall illustrate in an informal way the above process. Our aim is to present the concepts assuming just the basic knowledge of the execution of a program and basic mathematical concepts.

For understanding the basis of the method, let us take a look at the 'checkings' we do while doing calculations. Let us look at the checking of the following addition operation:

```
  1578                 1578
  8459                 8459
  6693                 6693
  2745                 2745
 ------               ------
 19475 (RESULT)        7255 (SUMS)
 ------                1222  (CARRIES)
                      ------
                      19475 (RESULT)
                      ------
   (A)                 (B)
```

Suppose, we have to check the addition done in (A); we have to *redo* the whole process of addition. Suppose we have to check the addition done in (B) where not only the summands of each position are given but also the carries to the immediate next position are given; it can be readily seen that the checking can be done easily. In the example given, there are no carries while adding SUMS and CARRIES. However, it is possible that there will be carries again while adding these in which case one can apply the process in a recursive manner. In the latter case if we assume that SUMS and CARRIES have been calculated *correctly* then the task of checking the addition of these two with the final result will become easy. Some of these principles are applied in establishing the correctness of programs.

Correctness of Programs

Using the simple programming language of *while-loops* discussed in the earlier articles of this series, we now illustrate the notion

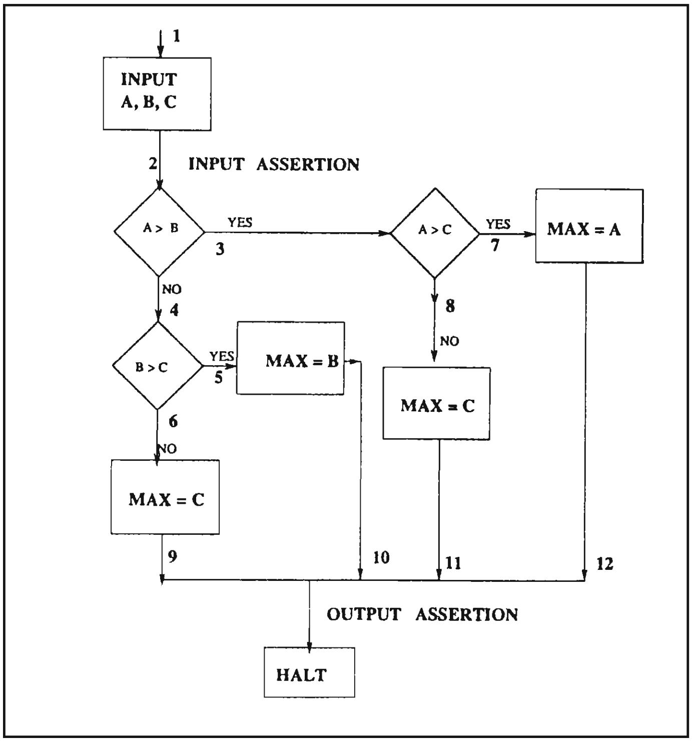


Figure 1. Flowchart for finding the maximum.

of *correctness* of programs. For ease of presentation, we use flowcharts rather than explicit while-programs.

As mentioned already we should now have a way of formalising the *intended behaviour*. One way of expressing the behaviour is to express the constraints on the inputs to a program and a relation between the input and the output variables. We express these constraints and relations in the following way:

One way of expressing the behaviour is to express the constraints on the inputs to a program and a relation between the input and the output variables.

1. **Input** assertions are of the form:

   ASSUME (<BOOLEAN RELATION>)

For example, the assertion ASSUME (X, Y Integers and $X > 0$, $Y > 0$) asserts that the values in X, Y are integers and the values of X and Y are both greater than zero. That is, we can assume that the values in X and Y satisfy these conditions just before the start of the execution of the program.

2. **Output** assertions are of the form:

   ACHIEVE (<BOOLEAN RELATION>)

- For example, ACHIEVE $(Z = X^Y)$ asserts that the value of Z is equal to X^Y at the end of the complete execution of the program.

- If we allow input variables also to change, we denote the original input values of the variables by the corresponding primed variables; for instance, if the input variables that can be altered are denoted X and Y then the corresponding variable names that contain their original values are denoted by X' and Y' Suppose we had written a program to swap the values of X and Y, then such an output assertion is denoted by ACHIEVE $(X = Y'$ and $Y=X')$

Example: Let us consider the simple program of finding the maximum out of three numbers X, Y and Z, the flowchart of which is shown in *Figure 1*. The input and output assertions of the flowchart are given by:

1. *Input assertion*: ASSUME (Distinct Integers $X, Y, Z > 0$)
2. *Output assertion*: ACHIEVE ($MAX \geq X, Y, Z$)

To show that the program satisfies the above *intended specifications*, we need to show:

- Each path starting from the edge labelled '2' (which is the place where the truth of input assertion is verified) [1] leads to a node where MAX is assigned and output assertion is satisfied. For instance, in the path 2-3-7 of *Figure 1*, we can see that at the edge labelled '3' we can assert using the meaning of the 'test' box that $A > B$ and the input assertion still holds; further at edge labelled '7', we can assert that $A > C$ and the assertion at node '3' is still valid. Using our *mathematical knowledge*, we can infer that $A > B, C$. From this assertion together with the meaning of the assignment statement, it can be easily seen that the output assertion follows.

[1] It must be noted that if the input assertion is not satisfied at this point, then any output assertion holds due to the classical implication operator.

- Successful analysis of the first step establishes that all paths starting from the input point to the output points satisfy the output assertion. If we can show that at each of the branching points along each of the paths all the possible cases have been considered, then from the success of step (1), we can assert that the program satisfies the intended specifications. In the flowcharts, 'tests' are essentially the branching points; thus at each of the nodes, corresponding to each of the possible answers (true/false or yes/no), we have to make sure that actions are specified. If there is a possible case which has been ignored then we say that there is a possibility of the program getting *stuck* or reaching a *dead-end*. In *Figure 1*, all the tests are binary tests and for each of the possible outcomes (yes/no), actions are specified. Hence, we can conclude that it is not possible for the program to get *stuck or reach a dead-end* at any point, and the correctness of the program follows.

Note: The impossibilities of certain cases (or certain input values) should necessarily be a part of input assertions. Such assumptions often lead to strengthening of the output assertions.

The basic idea of the above method lies in considering each node of the flowchart; from its input assertion and the meaning of the underlying command, derive the assertion that holds after the complete execution of the command corresponding to the node (possibly using the underlying theory). Usually, we shall refer to the assertion that is true before the control enters the node as *precondition* and the assertion that holds immediately after its complete execution the *postcondition* as highlighted in *Figure 2*. Apart from our knowledge of the execution of a program (in fact, a flowchart), we have used the following concepts:

1. *Mathematical knowledge*: We use this to mean the underlying theory; in the example considered, this can be interpreted to mean the theory of numbers (integers).

2. *The program can not get stuck*: This is interpreted to mean that

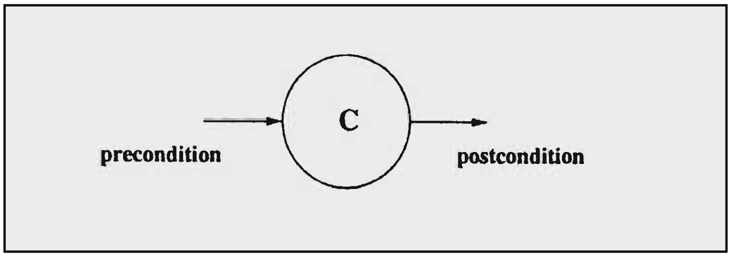


Figure 2. Assertions at a Node.

at every step the flowchart has definite actions to take till it reaches the *halt statement* for all intended possible values of the input variables.

We shall touch upon these aspects again subsequently. This assertion can be established from the fact that each of the 'tests' is completely covered and at the end of two 'tests' the relation among the three elements is fully established. The above analysis and illustration confines to flowcharts that are directed tree-like structures and hence, contain no cycles. Flowcharts with cycles correspond to programs with loops. One natural way of extending the previous approach to the class of programs with loops is to repeatedly unroll the loop once and apply the earlier verification procedure. That is, for each instantiation of the body of the loop, we apply the verification procedure once. Thus, the verification has to be applied as many times as the loop is executed. Thus, the proof will be different for each input value and further, we should know a priori how many times the loop gets executed. Obviously, the process does not end if the loop does not terminate. Such an approach is naturally undesirable. To overcome these drawbacks, we enrich the set of assumptions (and hence the specifications) by including an assertion for each loop called an *invariant assertion* having the form

INVARIANT(<BOOLEAN EXPRESSION>)

with the following interpretation:

An invariant assertion for a loop is true at the point of first entry and subsequent entries and holds immediately after the exit as well.

If a program has been shown to be correct relative to the input assumptions, the proof holds for all values satisfying the given input assertions.

Thus, the invariant assertion provides the needed induction for analysing programs that have potentially infinite number of paths. The method enables us to establish the correctness of the program for the whole domain of input. Thus, if a program has been shown to be correct relative to the input assumptions, the proof holds for all values satisfying the given input assertions.

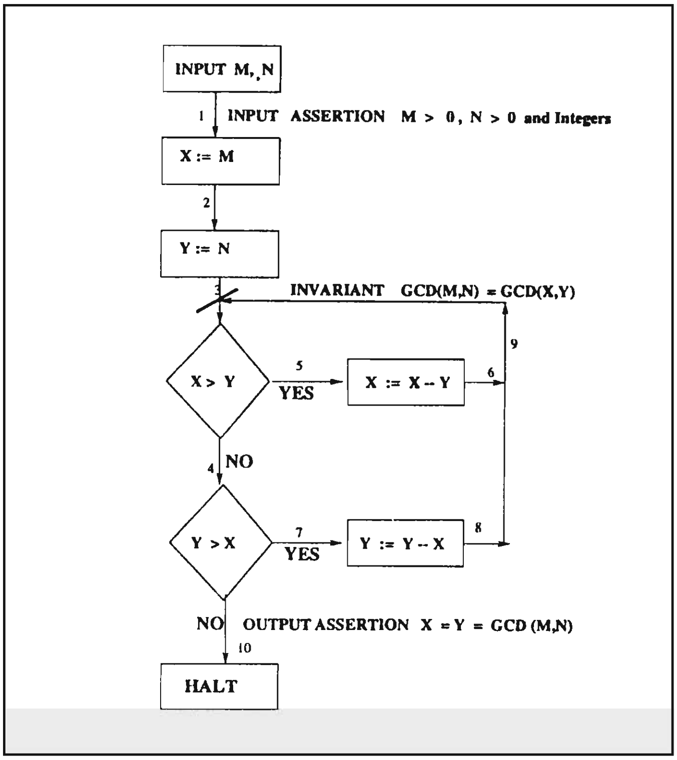

INPUT M, N
1
INPUT ASSERTION M > 0, N > 0 and Integers
X := M
2
Y := N
3
INVARIANT GCD(M,N) = GCD(X,Y)
9
X > Y
5
YES
X := X – Y
6
4
NO
Y > X
7
YES
Y := Y – X
8
NO
OUTPUT ASSERTION X = Y = GCD (M,N)
10
HALT

Figure 3. Flowchart for computing the GCD.

The method due to R W Floyd is referred to as the *Inductive Assertion* Method. For purposes of illustration let us consider the flowchart for computing greatest common divisor (GCD) of two numbers *M* and *N*. The assertions are marked in *Figure 3.*

Correctness: Before establishing the correctness, let us recollect some of the properties of GCD:

$$\begin{aligned}
&\text{(A1) } \mathrm{GCD}\,(x, y) = \mathrm{GCD}\,(y, x)\\
&\text{(A2) } \mathrm{GCD}\,(x, y) = \mathrm{GCD}\,(-x, y)\\
&\text{(A3) } \mathrm{GCD}\,(x, y) = \mathrm{GCD}\,(x+y, y) = \mathrm{GCD}(x-y, y)\\
&\qquad\qquad\qquad\;\; = \mathrm{GCD}\,(x, y+x) = \mathrm{GCD}(x, y-x)
\end{aligned}$$

Now, let us apply our earlier method to the example:

1. At edge 1, we have the input assertion.
2. The control enters the loop through edge 3. Here, we have to establish the invariant assertion. That is, we have to establish that the invariant holds on the control entering for the first time,

The method illustrated establishes that the program satisfies the intended behaviour captured through the various assertions.

subsequent entries, and at the exit of the loop. We can prove it as follows:

- On its first entry, assertion follows from the given assertions and the meaning of the assignment statements.
- For subsequent entries, we have to show that the invariant holds at edge 6 and edge 8. Now, since the 'tests' do not alter the values of the variables, it easily follows (or can be proved) that the assertion that is true at edge 3 also holds at edge 5 with the added assertion that $X > Y$. Using A3, we can now establish that the invariant holds on edge 6. Similarly the other points can be established.
- At edge 10, we can conclude that $X=Y$ from the meaning of the test commands. From our knowledge about inequalities, it follows that the invariant holds since the input assertion holding true at 3 continues to hold along edges 4 and 10 (argument similar to that given for edge 5).
- Again since the tests are binary, it follows from our knowledge about inequalities that all cases are considered, and hence the control does not reach any dead ends.

We have so far illustrated informally how correctness of programs are established. The method illustrated above establishes that the program satisfies the intended behaviour captured through the various assertions. To be a little more precise:

> Given the input and invariant assertions, the program satisfies the output assertions when it *terminates*.

It may be noted that the correctness is relative to the given specifications.

It may be noted that the correctness is relative to the given specifications. Further, in the correctness established we have not considered whether the program terminates or not. We use only the fact that when the program enters a node of a flowchart it exits that node in a finite amount of time. This only guarantees that the primitive commands take only a finite amount of time. This does not guarantee that the program terminates. In the case of the GCD program shown in *Figure 3*, the question of termination corresponds to whether the control entering through

It follows from Gödel's incompleteness theorem that any formal axiomatic system that satisfies certain minimum expressive requirements and is guaranteed to be consistent is doomed to be incomplete.

automatically. In the GCD example given above, the invariant is a property of GCD borrowed from the knowledge of Euclid's algorithm.

- We use almost at every step mathematical knowledge such as theory of numbers etc. It follows from Gödel's incompleteness theorem, touched upon in the previous article that any formal axiomatic system that satisfies certain minimum expressive requirements and is guaranteed to be consistent is doomed to be incomplete.

Termination of Programs: In the flowchart language all the primitive commands are terminating by definition. Hence, flowcharts without any loops, showing that the program can never get stuck or reach a dead-end, would imply termination. However, establishing *termination* of while-programs is non-trivial and is in general undecidable. In the following, we illustrate a technique that enables us to prove the termination of programs. Consider the GCD program shown in *Figure 3*. Basically, for establishing termination we have to show that starting from the input assertion, each loop can be executed only a finite number of times. From the loops of the program, it can be observed that for each execution of the loop either X is reduced or Y is reduced. Now, let us trace the value of $X+Y$, initially equal to $M+N$, for successive executions of the loop. Since $X > 0$ and $Y>0$, we can say

- $X+Y$ is reduced on each execution of the loop.
- From the tests, we can conclude that it remains positive and exits the loop when $X = Y$.

Since we have started from a finite number and each time the number is reduced by a positive quantity, we can conclude that the loop can be traversed only a finite number of times. From this, we can assert the termination of the program and conclude the total correctness of the program from the partial correctness of the program established already. Let us capture this a little more succinctly. Let $t(X, Y) = X + Y$. From our argument, it node 3 reaches the *halt* node or continues to execute the path 3-5-6-9-3 or 3-4-7-8-9-3 indefinitely. Furthermore, the notion of *termination* should not be confused with the possibility of the program reaching *dead-ends*. Thus, the following notions of correctness of programs are used in practice:

By showing that a program is partially correct we are sure that nothing bad happens.

1. ***Partial Correctness:*** We say that the program behaves as expected when it terminates. Thus, if the program does not terminate, we can as well prove anything as the control never comes to *halt* instruction. The illustrations above belong to this notion. By showing that a program is *partially correct* we are sure that *nothing bad happens*. That is why properties specified through such a characterisation are often called *safety* properties. It is interesting to note that several systems such as operating systems, real-time control systems are designed to work for ever, and hence by definition are *non-terminating*. The adaptation of partial correctness notions has enabled us to reason about such systems.

2. Total Correctness: We say that a program is totally correct if the partial correctness of the program and the termination of the program are established. By showing that the program is totally correct, we have guaranteed that *eventually something good happens*. Properties characterised through such specifications are often called *liveness* properties.

We say that a program is totally correct if the partial correctness of the program and the termination of the program are established. By showing that the program is totally correct, we have guaranteed that eventually something good happens.

In the above discussions, we have proved the correctness of a program in an informal language basing some of the inferences on our intuitive knowledge about the underlying theory. The above processes can be formalised in a logical framework without relying on the intuitive deductions we have used. The question then is:Does there exist an algorithm for establishing correctness? That is, does there exist an algorithm that can verify the correctness of any given program relative to its input and output assertions? The answer to this question is 'no' for the following reasons:

- The *invariant assertions* defined above cannot be obtained is clear that the sequence of values of t before the program halts will be a sequence of the form $X_0 = X+Y > X_1 > X_2 \ldots > X_n \geq 1$. In the domain of positive integers, it can be easily seen that the length of such sequences is bounded. The sets of values for such a sequence is given by $X_0, \ldots X_n$ for some bounded n. With the usual orderings on numbers such as '>', starting from any number we can have a finite chain of numbers satisfying the strict ordering '>' as there can be no infinite descending chain. Such sets are referred to as *well-founded sets* as there are no infinite descending chains starting from any element under the given order, >.

The general problem of termination of programs with loops is undecidable.

To sum up, the basic method of establishing termination lies in arriving at functions (of program variables) such as t whose sequence of values strictly decrease and do not form an infinite descending chain. The method is referred to as the *well-founded sets* method. A natural question that arises is: Does there exist an algorithm that can establish the termination or otherwise of any while-program? Again the answer is 'no'. In fact, the general problem of termination of programs with loops is undecidable. We shall illustrate this in the following subsection.

General Termination of Programs is Undecidable

Most proofs of undecidability are established using the classical method of *proof by contradiction* or reducing the problem to a known undecidable problem. In the method of proofs by contradiction, contradictions are resolved by questioning the assumptions made in the proofs. In the following, we provide a flavour for establishing the undecidability of the termination problem.

Most proofs of undecidability are established using the classical method of proof by contradiction or reducing the problem to a known undecidable problem.

Let us consider the well-known Russel's paradox formulated in terms of English adjectives referred to as Grelling's paradox.

> An adjective in the English language is said to be *autological* if it truly applies to itself. An adjective which is not autological is said to be *heterological.*

For example, *polysyllabic* has many syllables and hence by definition autological whereas *long* is not autological, and hence heterological. Let us ask the following question: Is *heterological* autological or heterological? The question can be answered in two ways:

1. Suppose we say that it is *autological.* Hence, by definition it applies to itself and thus, it must be heterological. This contradicts our answer.

2. Suppose we say that it is *heterological* and by definition, it leads to saying that it is autological – again contradicting our answer.

Thus, we derive contradictory answers either way. The conclusion of philosophers on such questions is that no answers are possible for such questions. Using programming notations, we can formulate the definition of heterological as given below:

heterological(x) = *not* (x(x))

Let us evaluate using this definition when 'x' is substituted by 'heterological'. We get

heterological(heterological) = *not* (heterological(heterological))

Thus, answering the left-hand side, reduces to evaluating the right-hand side, i. e. , *not* (heterological(heterological)). Now, we should evaluate 'heterological(heterological)' before we apply the *not* operator on the result. This goes on and on forever. In other words, we shall never get any response as suggested above. In the following, we show that assuming the existence of function that can check for the termination of a function (or a program) we can derive programs such as *heterological* above. Let us assume there is a function [2], *terminates*, which takes a function, say p, and its arguments, say x which answers the question whether $p(x)$ terminates or not. Using the *terminates* program we can program a function analogous to the *heterological* given above as follows:

2 The reader can understand *the use of function* in the sense of what is used in programming languages.

```
hetero (p) = if terminates (p,p) then not p(p)
             else true
             fi
```

The function *hetero* always terminates since (1) the function *terminates* terminates by assumption, (2) termination of $p(p)$ is ensured by the test corresponding to the then-clause and (3) the *not* operator is primitive and *true* is just a constant. Let us evaluate *hetero* with the argument *hetero.* We get

```
hetero (hetero) = if terminates (hetero, hetero) then not hetero(hetero)
                  else true
                  fi
```

Since the function *terminates* always terminates, from the above two equations we get:

hetero(hetero) = *not* hetero(hetero)

This is obviously a contradiction. Thus, we have to question our assumptions. Since assumptions (2)–(3) are trivially true, we question our assumption about the *terminates* function. The negation of this assumption corresponds to: It is impossible to program a function *terminates* that takes any program and its arguments and decides whether the program terminates or not.

It is impossible to program a function *terminates* that takes any program and its arguments and decides whether the program terminates or not.

Discussion

In this article, we have illustrated informally the notions of correctness of programs. The notions can be framed in a logical framework so that there is no need to call for intuition in order to recall the execution of programs.

Several formal frameworks exist for achieving the formalisation. Such methods are usually referred to as *formal methods*. In other words, formal methods are frameworks for reasoning about computational systems. Thus, they usually involve formal logic, mathematical structures and formal computer languages to formalise the specification (intended behaviour), and the verification of the program/system. In the previous sections, we

Formal Methods

- Formal methods are pivotal in the design, development, and maintenance of complex systems.
  - * Useful for establishing the completeness of the system and correctness with respect to its requirements.
  - * Useful for establishing correctness of the implementation to its higher level requirements.
  - * Various levels of formalism exist: formal specifications only, formal specifications with manual proofs/checkings, formal specifications with machine proofs.
- They are complementary to testing and form a good medium for review and reverse engineering. Formal methods cannot and should not be viewed as a substitute for identification and use of good abstractions relevant for the application domain.
- Formal methods can be applied to the whole system or only selected components; they can also be applied to specific aspects rather than for the full functionality of the system/component.
- Formal methods can be used at various stages of the lifecycle but are most cost-effective and have a bigger payoff when used in the earlier stages of development. They not only assist in the discovery of incomplete and ambiguous specifications, but also lead to clarification, crystallise incompatibilities in understanding of the clients/designers – thus resulting in better comprehension of the client's informal requirements and gains in assurance.
- It is possible to integrate formal methods to existing software engineering practices in a smooth manner – thus enhancing the confidence in software engineering methodologies.
Formal methods are most productive and effective when they are combined with other techniques such as prototyping, simulation, verification and testing.
- They are being used successfully on practical projects in industry.

have essentially illustrated a bottom-up approach to correctness; that is, verify (or check) a given program for its behaviour against its specifications. On the contrary, one can use the principles illustrated for developing *correct* programs starting from specifications. Such top-down methodologies have distinct advantages and are widely used in practice. In any formal analysis of systems, we generally deal with *models* of real systems for analysis. The problem due to aberrations between real and modelled world are the potential sources of errors; such problems are inherent in the application of any mathematical model. It is always necessary to gain confidence/validate the mathematical models used against real systems at least as far as the class of issues concerned. Thus, the correctness of systems in the most general sense are not usually achievable, and formal correctness concerns are always relative to the underlying formal models.

Suggested Reading

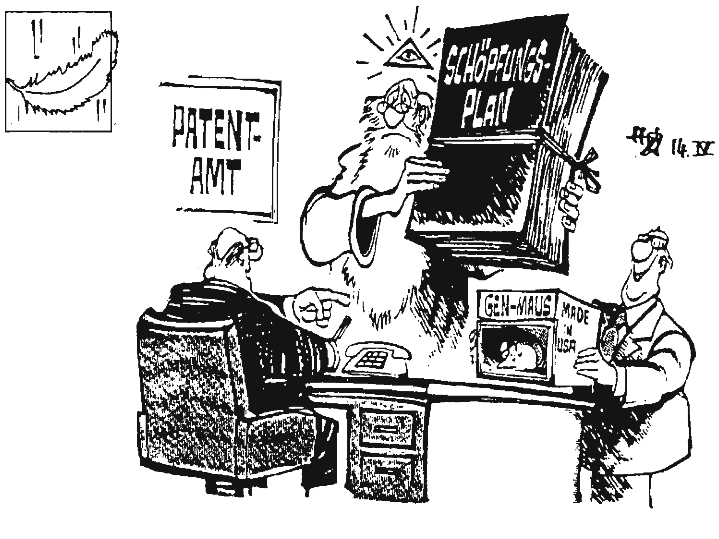
Correctness:

- N Wirth. *Systematic Programming: An Introduction*. Prentice Hall, Englewood Cliffs. New Jersey, 1972.
- E W Dijkstra. *A Discipline of Programming*. Prentice Hall. Inc., 1976.
- R G Dromey. *How to Solve it by Computer*. Prentice Hall International, 1982.

General and Undecidability:

- D Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley Publishing Co.. Inc., 1987.
- C A R Hoare and C B Jones. *Essays in Computing Science*. Prentice-Hall. Inc., 1989.
- C A R Hoare and D C S Allison. *Incomputability, ACM Computing Surveys*. September 1992.
- *Collected works of A M Turing*. North-Holland Publ. Co., Amsterdam, 1992.

Address for Correspondence
R K Shyamasundar
Computer Science Group
Tata Institute of Fundamental Research
Homi Bhabha Road
Mumbai 400 005, India
email:shyam@tcs.tifr.res.in
Fax:022-215 2110



On the book God is holding is written "Plan for Creation". The patent agent is saying, "I'm sorry, but this gentleman came before you!"

From: *Gene Antics*