Symbolic execution systems — a review

by P. David Coward

Symbolic execution is a technique that is useful in the validation of software. It may be used to aid in the generation of test data and in program proving. As software engineering becomes more concerned with the development of tools, symbolic execution will become an important item in the toolkit. This paper gives a review of symbolic execution and its applications. A minimum set of features for a symbolic execution testing system is set out. Of the 12 systems using symbolic execution only six contain these minimum features. These six systems are compared against an 'ideal' system. Based on this comparison some features for a new symbolic execution testing system are outlined.

1 Introduction

Software engineering is becoming more concerned with the development of tools to facilitate software development. This concern comes at a time when there is also an increasing interest in more formal methods of software development and verification. Symbolic execution will be a major component of these developments as it can be used to aid software testing and program proving. No other review of symbolic execution systems exists. This paper fills that gap by reviewing the systems described in the literature.

Symbolic execution, sometimes referred to as symbolic evaluation, does not execute a program in the traditional sense of the word. The notion of execution requires that a selection of paths through the program are exercised by a set of data values. A program which is executed using actual data results in the output of a series of values. In symbolic execution the data is replaced by symbolic values. A set of expressions, one expression per output variable, is produced.

The most common approach to symbolic execution is to perform an analysis of the program, resulting in the creation of a flow graph. This is a representation of a program which identifies the decision points and the assignments associated with each branch. By traversing the flow graph from an entry point, along a particular path, a list of assignment statements and branch predicates is produced.

The resulting path is represented by a series of input variables, condition predicates and assignment statements. Symbolic execution involves following this path from beginning to end. During this path traverse each input variable is given a symbol in place of an actual value. Thereafter, each assignment statement is evaluated so that it is expressed in terms of symbolic values of input variables and constants.

At the end of the symbolic execution of a path, the output variables will be represented by expressions in terms of symbolic values of input variables and constants. The output expressions will be subject to constraints. A list of these constraints, known as the path condition (PC), is provided by the set of symbolic representations of each condition predicate along the path. Analysis of these constraints may indicate that the path is not executable owing to a contradiction.

Consider path 1, 2, 3, 4, 6, 7, 9, 10, 11 through the program fragment shown in Fig. 1. The symbolic values of the variables and the path condition at each branch are given in the right-hand columns for the execution of this path.

In Fig. 1, in addition to the path condition, the symbolic expressions of X and R will be of greatest interest as these are the output variables.

The appendix, Section 9, contains a more detailed example of symbolic execution.

1.1 Difficulties facing symbolic execution

There are four areas which give rise to considerable difficulty for symbolic execution: the evaluation of loops; a dilemma over how to process module calls (calls to functions, procedures, subroutines and subprograms); the evaluation of array references dependent on input values; and checking the feasibility of paths.

The first problem concerns loops. Symbolic execution cannot proceed beyond a loop unless the number of iterations is known. When the number of iterations is dependent upon the input variables, determining the number of iterations requires the solution of recurrence relations. Such a solution, if derivable, is likely to yield a symbolic expression as opposed to an actual value.

The second problem involves module calls. The term 'module call' is used here to refer to the invocation of any out-of-line code. This includes subprograms that are compiled separately from the invoking program, internal subroutines, procedures and functions. The dilemma concerning module calls is whether to treat them using the macro-expansion approach (Refs. 1 and 2) or the lemma approach (Refs. 3 and 4).

For example, in a Cobol program, when the PERFORM verb is encountered during the symbolic execution of a path the execution may proceed by executing into the performed procedure. This is macro-expansion. Each time the module is invoked the symbolic execution will be repeated, starting anew at each invocation. The lemma approach, on the other

		Path condition	X	Y	Z	R
1	BEGIN	–	–	–	–	–
2	READ X, Y, Z, R	–	x	y	z	r
3	X := X + Y	–	x + y	y	z	r
4	IF X > Z	x + y < = z	x + y	y	z	r
5	THEN R := R + 1					
6	ENDIF	x + y < = z	x + y	y	z	r
7	IF Y = R	x + y < = z AND y < > r	x + y	y	z	r
8	THEN WRITE('Success', X, R)					
9	ELSE WRITE('Fail', X, R)	x + y < = z AND y < > r	x + y	y	z	r
10	ENDIF	x + y < = z AND y < > r	x + y	y	z	r
11	END	x + y < = z AND y < > r	x + y	y	z	r

Fig. 1 Program fragment and symbolic values for path 1, 2, 3, 4, 6, 7, 9, 10, 11

hand, symbolically executes a module once, and then uses the results each time the module is invoked.

The third problem concerns arrays. Many authors have identified arrays as problematic for symbolic execution (Refs. 2 and 5–8). There is no difficulty for an expression such as

A(5) := B

because A(5) is unique in the same sense that B is unique. Both refer to a specific single element, and this enables symbolic execution to proceed in the usual way. The difficulty arises when the index is an input variable, or is dependent upon at least one input variable, for example

A(I) := B

where I is an input variable. Symbolic execution cannot proceed in the usual way beyond this assignment, because the identification of the element within the array is not defined and is said to be an ambiguous array element. No approach has yet been identified which adequately solves the problem.

The fourth problem is the identification of infeasible paths. An infeasible path is one which cannot be executed owing to the contradiction of some of the predicates at conditional statements. It is surprising that, in a sample of programs, of the 1000 shortest paths only 18 were feasible (Ref. 9).

Consider the following block of code:

```
1   BEGIN
2     READLN (A);
3     IF  A > 15
4     THEN
5         B := B + 1
6     ELSE
7         C := C + 1;
8     IF  A < 10
9     THEN
10        D: D + 1
11  END;
```

There are four paths through this block, as follows:

- path 1 — lines 1, 2, 3, 4, 5, 8, 11
- path 2 — lines 1, 2, 3, 6, 7, 8, 9, 10, 11
- path 3 — lines 1, 2, 3, 6, 7, 8, 11
- path 4 — lines 1, 2, 3, 4, 5, 8, 9, 10, 11.

Path 1 can be executed so long as the value of A is greater than 15 after the execution of line 2. Path 2 can be executed so long as the value of A is less than 10 after the execution of line 2. Path 3 can be executed so long as the value of A lines in the range 10 to 15 inclusive after the execution of line 2. Path 4 cannot be executed regardless of the value of A because A cannot be both greater than 15 and less than 10 simultaneously. Hence this path is infeasible.

1.2 *Application of symbolic execution*

Symbolic execution has a variety of potential uses. It may be used in path domain checking, test data generation, program proving, program reduction and symbolic debugging.

Path domain checking:
When a path is executed with a single case it may result in:

- incorrect output owing to one or more faults (universally incorrect)
- correct output although a fault exists (coincidentally correct)
- correct output and no faults exist (universally correct).

To distinguish between a coincidentally correct output and a universally correct output a path must be executed by more than one case. The output from a symbolic execution is more general than an execution with actual data. It represents all the cases that could execute the path. Individual case values may be substituted into the symbolic expression and evaluated by hand. This may seem straightforward, but it is quite difficult to create test cases that satisfy all the constraints of a PC.

In principal, values may be chosen to exercise the path at extremes of the variable domains. Achieving such 'extreme' values will give greater confidence that values over the whole domain will be handled correctly. In practice, it may prove difficult to identify such test cases.

Test data generation:

The symbolic input values in the expressions produced for each output variable can be substituted with actual values. The values substituted constitute a test case and the evaluation of the expression provides the corresponding output value. The creation of such values may be automated by using a numerical optimiser. The PC is used as a set of constraints and the solution to an arbitrary objective function is used as a test case. This method of automatically generating comprehensive test data is likely to yield a smaller test set than other approaches, such as random testing.

Partition analysis:

Partition analysis is another technique that makes use of the output from symbolic execution to determine test data (Ref. 10). It uses symbolic execution to identify sub-domains of the input data domain. Here, symbolic execution is performed on both the program and the specification. An attempt is made to produce the sub-domains such that each sub-domain is treated identically by both the program and the specification. Where a part of the input domain cannot be allocated to such a sub-domain then either a structural or functional fault has been discovered in either the program or the specification.

This appears to be a powerful idea as it tackles the difficult task of reconciliation between the specification and the program. However, the specification must be expressed in a manner close to program source code. For this technique to prove practical, enhancements are required such that specifications can be expressed in a less program-like format.

Program proving:

The most widely reported approach to program proving is the 'inductive assertion verification' method after Floyd (Ref. 11). In this method assertions are placed at the beginning and end of selected procedures. 'A procedure is said to be correct (with respect to its input and output assertions) if the truth of its input assertion upon procedure entry ensures the truth of its output assertion upon procedure exit' (Ref. 4).

Assertions may be comparisons of variables such as at this point $A = B$ should be true, or a value should lie within a particular range, for example $A > 4$ AND $A < 10$. To determine whether these assertions hold the tester may insert them into the program. As symbolic execution proceeds the assertions are appended to the path condition.

When a path condition containing assertions is found to be feasible then the assertions on that path are consistent with the PC and the path is considered correct. Should a path condition become infeasible owing to the addition of an assertion then that assertion is inconsistent with the PC and the path is deemed incorrect.

This technique can be used to check the interaction between a program's component routines. Each routine is given a set of assertions. Infeasibility caused by a particular assertion pinpoints a problematic routine.

Program reduction:

King describes how symbolic execution can be used to achieve program reduction (Ref. 12). This is the act of taking a program and producing another program containing fewer statements. The result is 'a simpler program consistent with the original, but operating over a smaller domain' (Ref. 12). This is useful when re-using software where only a sub-set of the cases handled are required. A major step forward will have taken place in software engineering when the re-use of software is normal practice. Program reduction is a step towards this goal.

Symbolic debugging:

The tracing of the execution of a program is a powerful debugging technique. The more usual trace facilities display only the current value of the variables. A symbolic debugger enhances the tracing facilities by displaying the expressions for each variable, removing the need for by-hand walkthrough.

2 The minimum symbolic execution testing system

EXDAMS (Ref. 13) is a monitoring system constructed before the first symbolic execution systems. It requires that the program be executed with data values. While the program executes it is closely monitored and a history of the execution is stored. It provides a flexible approach to the examination of the execution history. An analysis of the source program is undertaken to provide a data table and model of the program. These are used in conjunction with the history to provide the user with a view of the program's state at any point during the execution.

After EXDAMS, many of the software testing and debugging tools made use of symbolic execution. Some of these systems incorporate features similar to those provided by EXDAMS. There are now 12 systems described in the literature which claim to use symbolic execution. These are EFFIGY (Refs. 6 and 14), SELECT (Ref. 1), ATTEST (Refs. 5 and 15), CASEGEN (Ref. 7), DISSECT (Refs. 6 and 16), EL1 Symbolic Executer (Refs. 3 and 17), SMOTL (Ref. 18), Interactive Programming System (Ref. 19), SADAT (Ref. 20), the Fortran Testbed (Ref. 2), IVTS (Ref. 21) and UNISEX (Ref. 22). The nature of the symbolic execution utilised varies from system to system. Additionally, SPADE (Ref. 23) and MALPAS (Ref. 24) are two commercially available systems which make use of path-based testing strategies where symbolic execution would be appropriate (Ref. 25).

The minimum feature that a system should exhibit before it can be considered to make use of symbolic execution may be as follows:

- It produces a path condition for each path examined.
- It determines whether a path condition is feasible.
- For each output variable it produces an expression in terms of input variables and constants.

Using these three features as criteria for rejection causes EXDAMS, DISSECT, EL1 Symbolic Executer, SMOTL, SADAT, IVTS and UNISEX to be removed from the list of symbolic execution systems.

EXDAMS has never been claimed as a symbolic execution system. Paths are executed with actual data and no symbolic expressions are maintained. Hence, no feasibility checking can be undertaken.

The DISSECT system produces expressions for variables and a PC for each path specified. It fails to check the consistency of the predicates of the PC: this task is left for the user to undertake by hand.

The EL1 Symbolic Executer does not create symbolic expressions for output variables, nor does it determine path feasibility.

SMOTL does not create expressions for variables nor maintain a PC. Its path analysis is based on maintaining maximum and minimum values for each variable. It uses these values to check for predicate contradiction and hence path feasibility. This is not symbolic execution but, nevertheless, it does have a strategy for combining paths in an attempt to reduce the number required to cover all branches.

SADAT does not attempt to determine path feasibility: it leaves this for the user to perform by hand. Expressions are not determined for output variables.

IVTS is still under development and it is the symbolic executor which is incomplete. As yet, the system does not determine path feasibility.

UNISEX is intended to use an expression simplifier and a theorem prover to assess path feasibility. In some cases the expression simplifier can resolve a PC to either true or false. When this cannot be achieved the resulting expression will be passed to the theorem prover. The theorem prover has not yet been built. Currently the PC is output for the user to assess by hand.

SPADE and MALPAS are systems which stem from work sponsored by the UK Ministry of Defence. There is little written material available which describes these systems or their inner workings and problems encountered in their development. As a result of this SPADE and MALPAS are excluded from this review.

This leaves six systems — EFFIGY, SELECT, ATTEST, CASEGEN, the Interactive Programming System and the Fortran Testbed — which satisfy the three criteria for use of symbolic execution. These systems can be considered to be symbolic execution testing systems.

3 Strengths and weaknesses of the systems

The improvements that should be incorporated into future symbolic execution systems and the weaknesses and omissions that should be overcome are identified for each of the six systems.

3.1 *EFFIGY — an interactive symbolic execution system*

EFFIGY was the first system to make use of symbolic execution. It was designed to allow a program to be developed and tested gradually, making use of the symbolic execution facility as well as execution with actual data values. It is a development tool which can be used when testing either program fragments as they are written, or complete programs.

The interactive nature of the system is one of its strengths since it allows a degree of flexibility. When a path is being evaluated the user may insert actual values as well as symbolic values. This results in expressions for some variables containing a mixture of symbolic and actual values. A symbolic trace of a path can be obtained. This shows the state of specified variables line by line.

A powerful feature of EFFIGY is the use of assert statements which can be verified, based upon the symbolic representations of the variables and PC.

Perhaps the most important ommission from the EFFIGY system, given that it is interactive, is the absence of a facility to provide a view of the flow graph. It is possible that a user could terminate a session with EFFIGY having discovered many faults, corrected them, and been satisfied with the modifications but having omitted some branches of the program. It would be helpful if an indication of the extent to which the analysis has covered the flow graph was provided. No long-term recording of coverage statistics appears to be maintained, although it is stated that an exhaustive 'test manager', of which no details are given, is a part of the system.

No strategy for path selection is incorporated into the system. The users must employ their own testing heuristics to create such a strategy. The selection of the appropriate branch, when this is not determined by the state of the PC, is left for the user; so too is any decision on the number of iterations to be made at each loop. Determining the feasibility of paths is achieved using a theorem prover.

There is no direct reference in the literature to how EFFIGY handles module calls. However, the lemma approach described by Hantler and King (Ref. 4) is thought to be the one incorporated in the system.

3.2 *SELECT — Symbolic Execution Language to Enable Comprehensive Testing*

SELECT (Ref. 1) was completed shortly after the EFFIGY system. SELECT attempts to provide similar facilities to EFFIGY and in addition the creation of values which can act as test cases. These values are generated as a by-product of evaluating path conditions. The system was built to process programs written in a sub-set of Lisp. It is not clear at what class of application the system is aimed.

SELECT produces three categories of output for each processed path:

- test data
- simplified symbolic expressions for each program variable on a path
- statement of correctness of user-supplied assertions.

SELECT also identifies some infeasible program paths.

The system aims to be automatic and is claimed to have a path selection strategy where the aim is coverage of every branch. Paths are created by commencing at the start of the program and adding one branch at a time until a halt is reached. Each time a branch is added to the PC, it is passed to an inequality solver to determine path feasibility. Both linear and non-linear inequality solvers are employed. The solver maximises an arbitrary objective function and the solution is used as a test case.

SELECT requires that, for loops which may be executed a variable number of times, the user specifies the number of iterations to be included in the path. The system updates the PC and variable expressions for each iteration of the loop.

Handling of subroutines is achieved by testing each subroutine in isolation until it is deemed satisfactory. Whenever such a subroutine is invoked, the set of path conditions for that routine may be incorporated into the path which invokes it. This adds to the combinatorial explosion of the number of possible paths. Boyer suggests that a better approach might be to make use of input and output assertions for each subroutine. This would not increase the number of paths (Ref. 1).

The ambiguous array reference problem, where arrays are indexed by input variables, is tackled by the introduction of virtual paths. This adds to the complexity of the flow graph by adding a branch for each element of the array.

3.3 *ATTEST — a system to generate test data and symbolically execute programs*

The prime objective of ATTEST is to generate test data for a path (Refs. 5 and 15). This is done using the conditions and assignments of the path to ensure that the data will force execution of that path. A secondary output, produced almost as a by-product, is a symbolic representation of the output variables of the path in terms of the input variables and constants. The system will sometimes inform the user if the path is infeasible and cannot therefore be executed because no such data set exists. The system cannot detect all infeasible paths. In particular, it cannot identify infeasibility where the system of constraints is non-linear.

ATTEST's main strength is significant. It analyses Fortran programs and hence was a step towards a widely usable symbolic execution testing system which can be used in either batch or interactive mode.

The difficult task of selecting the paths for analysis is not tackled by the system, but left entirely to the user. Paths for analysis can be specified in two ways: statically or interactively. The static specification requires the whole path to be specified at once, while the interactive mode allows the user to select one branch at a time as each conditional statement is reached.

Should a constraint make the path infeasible, the user is informed and the analysis proceeds to the next path. When the end of a path is reached and it is feasible then the final solution obtained is a test case that will cause execution of the path.

The system attempts to discover where array indexes stray out of bounds. It uses two temporary constraints. One specifies that the upper bound must be exceeded and the other that the lower bound must not be achieved. For example, consider an array A(5 … 10). Constraints of $I < 5$ and $I > 10$ are used. When applied separately each of these constraints should prove contradictory if the path will not allow the index to stray out of bounds. Apart from these array bounds constraints the system is weak in its processing of arrays. When ambiguous array elements are encountered the symbolic execution halts.

No path selection is undertaken by the system. The user must input the paths which are to be evaluated. When path constraints are checked for feasibility they are passed only to a linear inequality solver. The system analyses Fortran programs which might be expected to contain non-linear constraints. Without user intervention to check for linear constraints, path feasibility checking may be unreliable.

The processing of module calls uses macro-expansion. This does not cause a path explosion problem for the system because no recording of path coverage appears to be undertaken. It merely increases the length of the path. Recording of states, such as the PC and variable expressions, at each conditional are not maintained and there is no facility for insertion and verification of assertions.

3.4 *CASEGEN — an automated program test data generator*

CASEGEN generates test data and, like ATTEST, analyses Fortran programs (Ref. 7). The system operates entirely in batch mode and consists of four components:

- a Fortran source code processor
- a path generator
- a path constraint generator
- a test data generator.

The Fortran source code processor analyses the source and generates a database consisting of a flow graph, symbol table and an internal representation of the source code. The path generator uses the database to produce a set of paths to cover all branches. The authors of Ref. 7 do not discuss the path selection strategy, except to state that loops are executed a fixed number of times. For each path the path constraint generator produces the path condition. Some of the paths generated may be infeasible. The test data generator aims to create values for the input variables that satisfy the set of inequalities for each path, hence creating a test case for each path. The sets of inequalities in the path condition are solved using linear programming, integer programming, mixed programming and non-linear programming techniques as appropriate. A procedure based on systematic trial and error and random number generation is also used. This use of several types of optimiser makes this system superior to ATTEST in terms of feasibility checking. Nevertheless, it is error-prone. The numbers (26, 7, 7) were generated for the sides of an isosceles triangle (Ref. 7).

The builders of CASEGEN acknowledge the difficult aspects of symbolic execution. However, the difficulties of handling arrays are sidestepped rather than overcome. The ambiguous array reference is retained during symbolic execution and is resolved during test data generation. While the builders acknowledge the difficulties of handling module calls, they do not make clear what approach has been adopted in CASEGEN. Output of the variable expressions and PC are not provided, and there are no facilities for insertion and evaluation of assertions.

3.5 *Interactive Programming System*

The Interactive Programming System (Ref. 19) is a collection of integrated software support tools for the design, development and maintenance of large computer programs. It was built as a general software development tool for a language available on a small computer: MINIPL/1. It is a pity that a full version of a widely available language was not used. There is no statement in the literature concerning the success of the system in the field, but it was built in conjunction with Olivetti and was destined for commercial use.

The system translates the source program into an intermediate representation. All subsequent output messages refer to the intermediate representation. The user must undertake, by hand, the task of relating the output, in terms of the intermediate representation, to the source program. This would not be acceptable to many users even if the differences between the source and intermediate representations were cosmetic, such as the breaking of statements over larger or smaller numbers of lines. In this case many constructs are different, making the output unacceptable.

The interactive part of the system is concerned with the direction of the symbolic execution. Its strengths centre around the state recording and review facilities. Whenever a conditional statement is encountered the user is required to select the branch to be pursued. The state of the PC and variables at each conditional statement are recorded. The user may return to any previously encountered conditional statement and examine the states of PC and variables.

Assertions may be introduced into the program. The system will determine the consistency between the assertion and the path condition. The flexibility provided by the interactive nature of the system is used to overcome the absence of a path selection strategy. While arrays and calls are handled the mechanisms employed are not made clear.

3.6 Fortran Testbed

The Testbed (Refs. 2 and 26) undertakes static and dynamic analysis of Fortran programs, construction of paths and generation of test cases which will cause the execution of the paths. It is the path construction and test case generation that is of concern here as these make use of symbolic execution.

The Testbed makes use of the LCSAJ construct (Ref. 27). An LCSAJ (linear code sequence and jump) is a series of statements ending with a transfer of control out of the linear code sequence. Paths can be viewed as a series of LCSAJs. Determining whether each LCSAJ is feasible can be the first step in determining the feasibility of a path. Should a single LCSAJ be infeasible then any path of which it is a component is infeasible. A path whose LCSAJs are all feasible may not itself be feasible. Two LCSAJs together may give rise to infeasibility. Gradually LCSAJs may be added together until either the addition of an LCSAJ results in the creation of an infeasible path or the path is complete and feasible. The Fortran Testbed determines the feasibility of an LCSAJ or a series of LCSAJs by employing symbolic execution. The path selection strategy does not deal with loop iterations but leaves this for the user to specify.

Having identified a feasible path the system goes on to generate test data which will cause the execution of the path. The system will solve sets of linear inequalities, provided by the path condition, to produce test data. The system makes use of random number generation to solve non-linear inequalities. This approach is similar to that employed by CASEGEN. The Fortran Testbed has an additional feature which makes use of an algorithm to establish bounds for the random number generation. It is based on the constants in the inequalities. In a series of tests the system solved and generated test data for paths that covered all but 2.7% of LCSAJs.

The system analyses Fortran programs and, with a few minor exceptions, this is ANSI standard Fortran. Symbolic execution is used as one technique employed by a software testing system that provides a variety of testing facilities such as static analysis through to test case generation. The Testbed is a commercially used system and cannot be regarded as just a researchers' experimental system.

Most module calls are handled by macro-expansion. This increases the number of paths available for consideration. Functions, however, are treated as input statements and new symbolic values are provided. It is not clear why different strategies are employed for function calls and other calls. Treating both as input/output interfaces would maintain the boundary identified during design.

The Testbed does not cater for the inclusion of assertions.

4 The ideal symbolic execution testing system

The notion of an ideal system ignores the practical considerations of construction. A system that delivers the features of an ideal system but, for example, with lengthy processing time is clearly not ideal. Requirements of an ideal system are likely to be contradictory, and a practical system must be a compromise between the conflicting requirements. Nevertheless, it is useful to set out an optimist's system against which existing and potential systems may be compared.

4.1 Input

The primary input to such a system is the source program, which may contain assertions. Further inputs will be required in response to the results obtained:

- a specification of paths to be evaluated
- changes to output assertions
- changes to input assertions
- a simple mechanism for the input of the expected results for generated test data, either as a range of values or a single value.

4.2 Output

The power of a symbolic execution testing system is determined by the variety and utility of the information provided. This should include:

- a diagrammatic representation of the program flow graph
- paths (list of constituent statements)
- the PC and an indication of feasibility or the constraint which turns the path infeasible
- expressions for output and intermediate variables
- display idle or active domains for input variables
- statements on truth of each assertion
- test data
- results of execution
- comparison of results versus expectation
- a statement on the coverage obtained in the testing already undertaken.

The utility of the output may be enhanced by providing a variety of viewing modes:

- a display of idle and active domains for intermediate variables on particular paths
- a display of specified variables or the PC on specified paths at specified points

- a trace through the symbolic execution of a path at various speeds
- a trace through the execution of user-provided test cases
- a trace through both symbolic and test case execution simultaneously
- a request facility for a coverage report at any stage of testing.

4.3 Path selection

Current methods of path selection employ only simple strategies, such as take the true branch first, generate the shortest path. These strategies have a common target: that of achieving a particular coverage metric, such as all statements or all branches or all LCSAJs are executed at least once. Each of these strategies is prey to the problem of selecting infeasible paths. When a set of paths which cover, say, all branches have been identified, some of the selected paths will be found to be infeasible. This leaves the problem of identifying feasible paths which include the non-covered branches from the infeasible paths. This strategy does not consider the effectiveness of the paths selected and whether the paths are in some way more useful or interesting than other paths.

An alternative approach may be to attempt to identify paths and associated test cases that are, in some way, representative of a large set of cases. The ideal symbolic execution testing system should incorporate a path selection strategy in which the expressions produced by the symbolic execution are utilised in an attempt to identify 'interesting' paths. Further work in this area is needed.

The ideal system also requires novel solutions to the problems presented by loops, arrays and module calls.

4.4 A multi-language symbolic execution system

Researchers have reflected on the possibility of producing a general symbolic execution system which may be used regardless of the language in which the source program is written. Such a system should exhibit two necessary features. First, there is a need for a translator from each source language into the single intermediate representation processed by the evaluator. The intermediate representation must therefore cater for all features of the set of source languages. Secondly, output messages from the execution system should be meaningful to the user. For example, when path infeasibility has been detected by the system the output should clearly identify the path and the predicate which turned the PC infeasible. The most instructive format for this output is where it refers to the source program submitted to the system. To achieve this requires the maintenance of references to the original source program for look-up before output of messages.

An alternative, more realistic, target may be to require that a symbolic execution system should analyse programs written in a widely used language. For example, no symbolic execution system has been built for a commercial data processing language such as Cobol. Whatever the language, it should also handle all the features of that language. This is no straightforward matter given the large number of subsets and super-sets installed by individual manufacturers. Nevertheless, adherence to a widely adopted standard, such as an ANSI standard, would be useful. It would be similarly prudent for the system to be built in a widely used language, allowing ease of installation of the system.

5 Ideal, existing and new symbolic execution systems

Table 1 summarises the features of the ideal and the six systems described above.

The first system to be built that satisfied the minimum criteria for a symbolic execution system was EFFIGY. It is also EFFIGY that is closest to the ideal system. All subsequent systems were, in some senses, a step backwards from the standard set by EFFIGY.

The only major weakness of EFFIGY when compared with later systems is its failure to adopt a widespread language. The authors do not state the language in which it is written but more important is the fact that it handles only a simple subset of PL/1 — not very useful. If the other systems were assessed in terms of their ability to handle similarly simple language sub-sets there is a possibility that they may appear as impressive as EFFIGY.

In a bid to make them more useful many of the subsequent systems handle larger language sub-sets and more popular languages. This has been achieved at the expense of a reduction in the capabilities of the systems. The system that is closest to EFFIGY in capability yet handles a complete and popular language is the Fortran Testbed. If this system were to incorporate a mechanism for the specification of assertions then this would become closer to the ideal system than the EFFIGY system.

Assertions could, of course, be written as normal source code. This would mean that following testing the program would be in need of modification to remove the assertions. It could be argued that permanent assertions with appropriate error messages could provide a powerful run time semantic monitor.

EFFIGY is the system against which new symbolic execution systems should be measured. If a new system will not provide the features provided by EFFIGY then it is unlikely to be worth constructing. The minimum features necessary to be comparable with EFFIGY are:

- input of source code containing assertions
- interactive input of paths and variables to be displayed
- output of:
  - ☐ symbolic values
  - ☐ truth of assertions
  - ☐ statement on path feasibility.

Additional features necessary to justify the creation of a new system are:

- a widely used language (ideally not accommodated by current systems)
- a path selection strategy.

It is the development of a path development strategy that is fundamental to the advancement of symbolic execution testing systems and other forms of path-based testing.

6 Summary

Symbolic execution is a path-based technique that employs symbols in place of data values. It provides expressions for output variables in terms of input variables and constants, and a set of constraints, the path condition, that must be

Table 1 Features of symbolic execution systems

System	Date	Language built in	Language analysed	Batch/ interactive	Input	Output	Handles calls	Handles loops	Handles arrays	Path selection	Path feasibility	Assertion testing
Ideal system		widely available	many widely available languages	batch and interactive processing	source program; assertions	path; PC; point becomes infeasible; output variable expressions; truth of assertions; coverage details; test data; active/idle domains	a solution where the problem of path explosion is avoided yet the coverage is complete			uses symbolic expressions to aid selection	linear and non-linear	yes
EFFIGY	1975	not stated	simple PL/1	interactive	assertions; display locations	symbolic values; truth of assertion	yes, lemma	user selects	one-dim	user selected	indicated at branch where becomes infeasible; theorem prover	yes
SELECT	1975	Lisp	Lisp subset	combination	assertions	not stated	yes, macro-expansion	user specifies maximum number of iterations	yes	all paths!	at point becomes infeasible; algebraic, linear and non-linear	yes
ATTEST	1976	Fortran	Fortran	batch or interactive	paths	symbolic expressions; test cases	yes, macro-expansion	fixed maximum number of iterations	only constant indexes	none	at point becomes infeasible; algebraic, only linear	no
CASEGEN	1976	Fortran	Fortran	batch	source program	path; test cases	not stated	fixed number	yes	minimal set covers all branches	at point becomes infeasible; algebraic, linear, non-linear and random	no
IPS	1979	PL/1	Subset of PL/1	interactive	source program; assertions; branch selections	symbolic expressions; path conditions; state of assertions	not stated	user selects	not known	user control	when requested by user; theorem prover	yes
Fortran Testbed	1981	Algol 68 Fortran	Fortran	batch	source program	paths; test data	yes, macro-expansion; functions as input	user specified	yes	not stated	at point becomes infeasible; algebraic, linear and random	no

satisfied for a path to be executed. By solving the path condition a test case is generated that causes execution of the path.

Eleven systems have been built which make use of this technique, but only six of them adopt the three core features of symbolic execution: production of a path condition; determining the feasibility of a path condition; and establishing an expression for each output variable. Of these six systems EFFIGY, ironically the first to be built, appears to be the best foundation for the development of future symbolic execution testing systems.

The greatest problem facing the development of symbolic execution testing systems, and any other path-based testing technique, concerns the selection of paths.

7 Acknowledgments

The author wishes to thank Prof. Darrel Ince and Prof. Mike Hennell for their comments on early drafts of the paper. Thanks are also due to the referees for their helpful suggestions.

8 References

1 BOYER, R.S., ELPAS, B., and LEVIT, K.N.: 'SELECT — a formal system for testing and debugging programs by symbolic execution'. Proceedings of International Conference on Reliable Software, April 1975, pp. 234–244

2 HEDLEY, D.: 'Automatic test data generation and related topics'. Ph.D. Thesis, Liverpool University, Liverpool, England, Aug. 1981

3 CHEATHAM, T.E., HOLLOWAY, G.H., and TOWNLEY, J.A.: 'Symbolic execution and the analysis of programs', *IEEE Transactions on Software Engineering*, 1979, **SE-5,** (4), pp. 402–417

4 HANTLER, S.L., and KING, J.C.: 'An introduction to proving the correctness of programs', *Computing Surveys*, 1976, **18,** (3), pp. 331–353

5 CLARKE, L.A.: 'A system to generate test data and symbolically execute programs', *IEEE Transactions on Software Engineering*, 1976, **SE-2,** (3), pp. 215–222

6 KING, J.C.: 'Symbolic execution and program testing', *Communications of ACM*, 1976, **19,** (7), pp. 385–394

7 RAMAMOORTHY, C.V., HO, S.F., and CHEN, W.J.: 'On the automated generation of program test data', *IEEE Transactions on Software Engineering*, 1976, **SE-2,** (4), pp. 293–300

8 HOWDEN, W.: 'Symbolic testing and the DISSECT symbolic execution system', *IEEE Transactions on Software Engineering*, 1977, **SE-3,** (4), pp. 266–278

9 HEDLEY, D., and HENNELL, M.A.: 'The cause and effects of infeasible paths in computer programs'. Proceedings of Eighth International Conference on Software Engineering, London, England, 1985

10 RICHARDSON, D.J., and CLARKE, L.A.: 'A partition analysis method to increase program reliability'. Proceedings of Fifth International Conference on Software Engineering, San Diego, CA, USA, 1981, pp. 244–253

11 FLOYD, R.W.: 'Assigning meaning to programs'. Proceedings of Symposia in Applied Mathematics 19, 1967, pp. 19–32

12 KING, J.C.: 'Program reduction using symbolic execution', *SIGSOFT Software Engineering Notes*, 1981, **6,** (1), pp. 9–14

13 BALZER, R.M.: 'EXDAMS Extendable Debugging and Monitoring System'. AFIPS Conference Proceedings of Spring Joint Computer Conference, Boston, MA, USA, May 1969, pp. 567–580

14 KING, J.C.: 'A new approach to program testing'. Proceedings of International Conference on Reliable Software, April 1975, pp. 228–233

15 CLARKE, L.A.: 'A program testing system'. Proceedings of Annual Conference of ACM, Houston, TX, USA, Oct. 1976, pp. 488–491

16 HOWDEN, W.: 'DISSECT — a symbolic execution and program testing system', *IEEE Transactions on Software Engineering*, 1978, **SE-4,** (1), pp. 70–73

17 PLOEDEREDER, E.: 'Pragmatic techniques for program analysis and verification'. Proceedings of Fourth International Conference on Software Engineering, Munich, West Germany, Sept. 1979, pp. 63–72

18 BICEVSKIS, J., BORZOVS, J., STRAUJUMS, U., ZARINS, A., and MILLER, E.F.: 'SMOTL — a system to construct samples for data processing program debugging', *IEEE Transactions on Software Engineering*, 1979, **SE-5,** (1), pp. 60–66

19 ASIRELLI, P., DEGANO, P., LEVI, G., MARTELLI, A., MONTANARI, U., PACINI, G., SIROVICH, F., and TURINI, F.: 'A flexible environment for program development based on a symbolic interpreter'. Proceedings of Fourth International Conference on Software Engineering, Munich, West Germany, Sept. 1979, pp. 251–263

20 VOGES, V., GMEINER, L., and AMSCHLER, A.: 'SADAT — an automated testing tool', *IEEE Transactions on Software Engineering*, 1980, **SE-6,** (3), pp. 286–290

21 TAYLOR, R.N.: 'An integrated verification and testing environment', *Software — Practice & Experience*, 1983, **13,** pp. 697–713

22 KEMMERER, R.A., and ECKMANN, S.T.: 'UNISEX: a Unix-based symbolic executor for Pascal', *Software — Practice & Experience*, 1985, **15,** (5), pp. 439–458

23 SPADE. Program Validation Ltd., 34 Bassett Crescent East, Southampton, England

24 'The capabilities of MALPAS, a software verification and validation tool'. RTP/4002, Rex, Thompson and Partners Ltd., Farnham, Surrey, England, April 1987

25 CLUTTERBUCK, D.L., and CARRÉ, B.A.: 'The verification of low-level code', *Software Engineering Journal*, 1988, **3,** (3), pp. 97–111

26 HENNELL, M.A., HEDLEY, D., and RIDDELL, I.J.: 'The LDRA software testbeds: their roles and capabilities'. Proceedings of IEEE Software Fair 83 Conference, Arlington, VA, USA, July 1983

27 HENNELL, M.A., WOODWARD, M.R., and HEDLEY, D.: 'On program analysis', *Information Processing Letters*, 1976, **5,** (5), pp. 136–140

P.D. Coward is with the Department of Computer Studies, Bristol Polytechnic, Coldharbour Lane, Frenchay, Bristol BS16 1QY, England, and the Computing Discipline, Faculty of Mathematics, The Open University, Walton Hall, Milton Keynes MK7 6AA, England.

9 Appendix — symbolic execution of a path through 'the triangle program'

Before considering symbolic execution of a path it is useful to introduce the directed graph as a representation of a program. It is easier for the reader to identify a particular path from a directed graph than from the source code of the program. A directed graph consists of *nodes* joined by *arcs*. The nodes represent branch points within the program and the arcs represent the statements that modify variables.

The program shown in Fig. 2 accepts three integers that are interpreted as lengths of the sides of a triangle. It determines the type of triangle that the values represent and calculates the area of the triangle.

The program can be represented by the directed graph in Fig. 3. A node on the graph corresponds to a selection statement in the program and an arc on the graph corresponds

```
BEGIN
INPUT I, J, K
IF I + J > K AND J + K > I AND K + I > J
  THEN Match := 0
  ELSE PRINT 'Not a Triangle'
       STOP
ENDIF
S := (I + J + K)/2
A := S–I
B := S–J
C := S–K
Area := (S*A*B*C)**0.5
IF I = J
  THEN Match := Match + 1
ENDIF
IF J = K
  THEN Match := Match + 1
ENDIF
IF K = I
  THEN Match := Match + 1
ENDIF
CASE OF MATCH
                = 0 THEN PRINT 'Scalene'
                = 1 THEN PRINT 'Isosceles'
                = 3 THEN PRINT 'Equilateral'
     OTHER THEN PRINT 'Error'
ENDCASE
PRINT Area
END
```

Fig. 2 The triangle program

to a series of input, assignment and output statements.

The directed graph contains 33 paths. Symbolic execution considers a single path at a time.

The symbolic execution commences at the root of the path and proceeds, arc by arc (branch by branch), to the end (leaf) of the path. The branches can be written in sequence to isolate the path. The expressions in square parentheses are the predicates that must be true for execution of the path to take place. Consider the path covering branches 1, 2, 4, 6, 8, 12 and 14 as follows:

```
1   INPUT I, J, K
2   [I + J > K AND J + K > I AND K + I > J]
    Match := 0
    S := (I + J + K)/2
    A := S–I
    B := S–J
    C = S–K
    Area := (S*A*B*C)**0.5
4   [I = J]
    Match := Match + 1
6   [J = K]
    Match := Match + 1
8   [K = I]
    Match := Match + 1
12  [Match = 3]
    OUTPUT 'Equilateral'
14  OUTPUT Area
```

Symbolic execution of this path is now described:

1 SET I to i
SET J to j
Set K to k

2 To progress beyond this point the condition enclosed within the square parentheses must be satisfied. The test case must satisfy this condition if the path is to be executed. (The condition that must be satisfied to cause execution of a path is termed the path condition (PC).)

PC is i + j > k AND j + k > i AND k + i > j
Set Match to 0

When a variable is set to a numeric value the operation is carried out as part of the symbolic execution.

Set S to (i + j + k)/2

When a variable is assigned the value of an expression the variables that are incorporated in the expression are substituted by their current symbolic expression. In the case of I, J and K their symbolic values at this point are i, j and k, respectively.

Set A to (i + j + k)/2–i

This symbolic value of A is achieved by substituting S and I with their symbolic values.

Set B to (i + j + k)/2–j
Set C to (i + j + k)/2–k
Set Area to (((i + j + k)/2)*((i + j + k)/2–i)
((i + j + k)/2–j)((i + j + k)/2–k))**0.5

4 In addition to the PC constructed at step 2 an additional condition is now imposed on the path. The PC now becomes:

PC is i + j > k AND j + k > i AND k + i > j AND i = j

The PC may be simplified to an equivalent PC.

PC is i = j AND 2*i > k AND k > 0
Match = 1

Match is incremented by 1.

6 j = k is a further constraint so the path condition now becomes:

PC is i = j AND 2*i > k AND k > 0 AND j = k

Simplifying gives:

PC is i = j AND j = k AND k > 0
Match = 2

8 k = i is a further constraint which is added to the PC.

PC is i = j AND j = k AND k > 0 AND k = i

Simplifying merely removes the added constraint k = i to give:

PC is i = j AND j = k AND k > 0
Match = 3

12 PC is i = j AND j = k AND k > 0 AND Match = 3

14 PC is i = j AND j = k AND k > 0 AND Match = 3

Branch 14 does not cause a change to a variable nor to the PC.

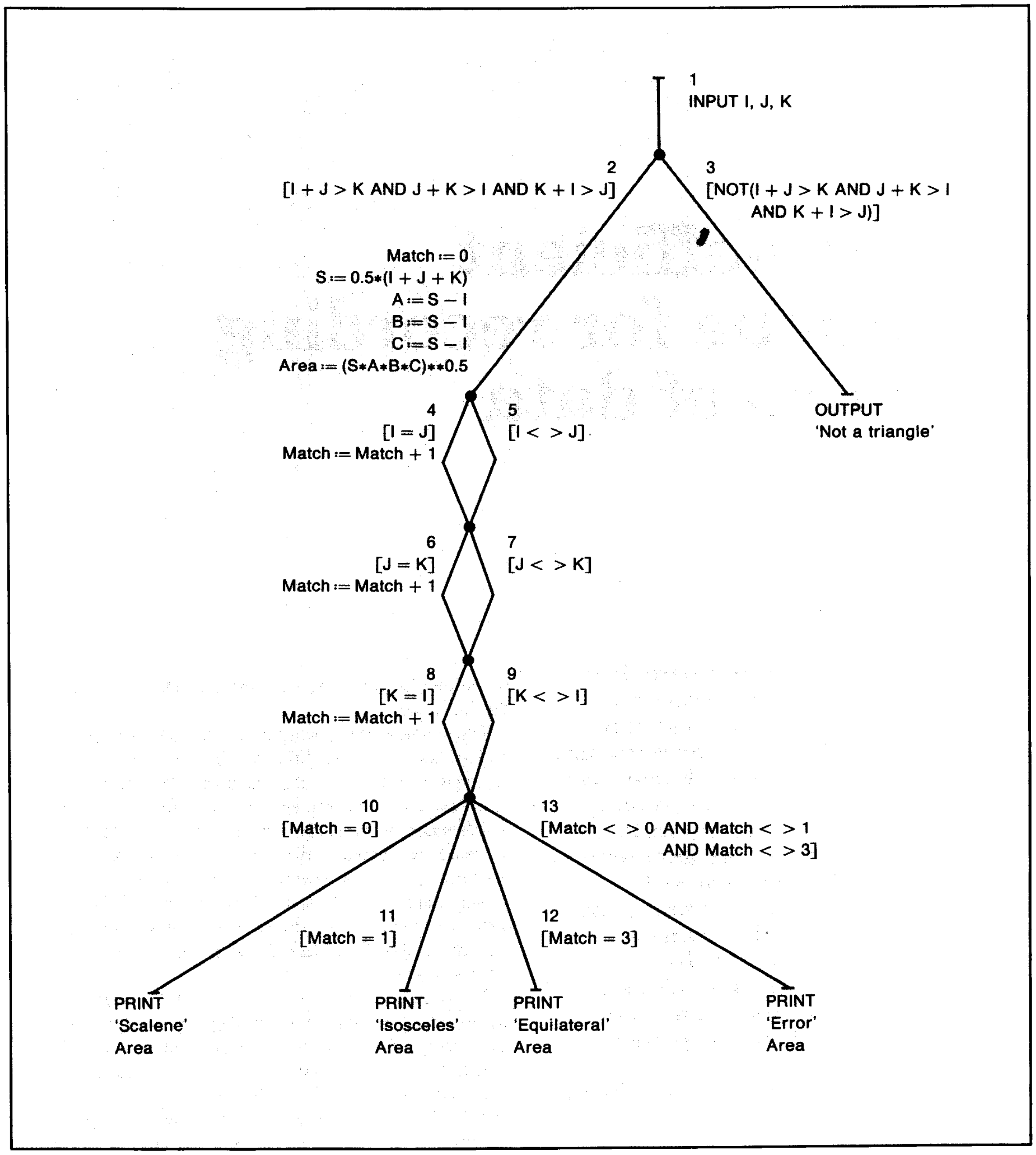

Fig. 3 The directed graph of the triangle program

For this path the following inputs and outputs exist:

Inputs
I = i
J = j
K = k
Outputs
'Equilateral'
Area = (((i + j + k)/2)*((i + j + k)/2–i)
((i + j + k)/2–j)((i + j + k)/2–k))**0.5
PC is i = j AND j = k AND k > 0

Note there is no reference to the variable Match in the output path condition. Only constraints for input variables in terms of constants and other input variables are included as it is only input variables which can be varied. As a result the constraint of Match = 3 is not present.

The path condition clearly shows that output of the string 'Equilateral' requires i = j = k > 0, which matches with expectation. It is not so straightforward to verify that the output for Area is correct. This is most easily checked by using the PC as a set of constraints in the minimisation of an arbitrary objective function. When solved, the values provide a test case which may be used to execute the path and hence evaluate the expression for Area.