Transactions on Information and Communications Technologies vol 9, © 1994 WIT Press, www.witpress.com, ISSN 1743-3517

Generation of software tests from specifications

I. Spence & C. Meudec

Department of Computer Science, The Queen's University of Belfast, University Road, Belfast, BT7 1NN, Northern Ireland, UK

ABSTRACT

Thorough testing is widely acknowledged to be a very expensive part of the software development process. The conventional method of constructing and executing tests for software systems is contrasted with automatic techniques which generate and execute tests derived from the software under test and/or from its formal specification. We present a review of techniques which are currently being used or developed for generating tests, and discuss the approach which we are using.

INTRODUCTION

In spite of the greatest care being taken during the specification, design and development of any product it is rarely disputed that confidence in the end result can and should be increased by trial use prior to general release and widespread use. For software systems this trial use, or testing, can serve two purposes, depending on who carries it out -

- A software engineer who should if possible not have been involved with the development of the software might carry out testing to ensure that the specification has been implemented correctly, that is, that the software does what the developers thought it would do.
- A user might carry out testing to ensure that the software does what the users thought it would do, that is, test the specification as well as the software.

In this paper we only consider the former kind of testing and so we assume that the expected behaviour of the software is precisely written down.

Testing involves generating artificial data, executing the program with this data, and observing the results. The data for a single execution, together with some means of determining whether the corresponding results are acceptable, is called a *test case.*

The traditional approach involves manual construction and execution of tests and inspection of results. Someone carefully reads the specification and/or the implementation of a module or program, and invents tests which exercise the program as thoroughly as possible. The construction and execution of these tests takes time and is therefore expensive, both in terms of human resources and in terms of costs introduced by delayed program release. It is commonly understood, for example, that approximately 30% of the costs incurred during software development can be related directly to testing, so there is considerable pressure to use test sets which are as small and as efficient as possible. In many respects manual testing is a very tedious process, involving prolonged attention to fine detail and repetition of similar tasks, frequently without significant intellectual challenge. The tests generated in this way are very subjective. Many researchers have therefore investigated the possibility of automating different parts of testing.

Automatic execution of tests

Depending on the exact nature of the interfaces to the program, automation of the execution of tests can frequently be straightforward. If all input to the program is from files on disc then test data can be prepared in advance and supplied to the program automatically. This technique can be used to ensure that the same tests can be carried out on a new version of the software to ensure that any modifications have not destroyed old capabilities (useful for regression testing).

If a program relies on input from a user via a keyboard and mouse, as is increasingly common, the preparation and use of data files is more difficult. The technique used is to introduce a layer of testing software between the user and the program under test. The user carries out the tests once, and the test system records keystrokes and mouse activity. These can then be reproduced on a subsequent occasion to simulate the activity of a real user, reducing the time taken for subsequent test runs and ensuring that the tests are reproduced accurately.

Automatic interpretation of results

Having supplied the data to the program under test it is necessary to decide whether the results produced are acceptable. To achieve this the corresponding output must be available to the test system and there must be some way for the test system to determine whether the output is acceptable. As before, the degree of difficulty is affected by the nature of the interface. If the only output is to disc files, is may be a straightforward matter to decide whether this is acceptable, for example by comparing it against a previously determined correct version.

For graphical output to a high-resolution screen however there is no easy way for a test system to ensure that the output is acceptable. For example, a shift in the position of a screen item by one pixel may well not have any visual significance, but would be rejected by the simplistic approach of recording each pixel written to the screen and comparing against some agreed version. This is an instance of the more general situation where the program specification is deliberately non-deterministic, that is where there is more than one acceptable solution. With the rapid proliferation of programs

which are event-driven and which have graphical user interfaces, this is a growing problem.

In spite of the problems mentioned above, considerable use has been made of these techniques for the automatic recording, playback and interpretation of results. Significant time savings can be achieved and there is increased confidence that tests can be repeated exactly. In environments where software development is audited this is regarded as being of increasing importance.

Automatic generation of test cases

Automating the generation of test cases, together with verification of the corresponding results, has undoubtedly proved to be the most difficult of the three tasks. It requires that the test generation system be able to analyse the program specification and, possibly, the program implementation, and exercise as much of both as possible.

Tests can be generated on the basis of the specification alone, which leads to black-box testing - the program under test is considered to be opaque and it is not possible to know anything about it except by supplying it with input and examining the output. This has the advantages that the tests can be generated before the program has been implemented, and that the same tests can be applied to different implementations. Making use of the specification also means that it is possible to determine whether the output produced by the program is acceptable.

It is also possible to construct tests which are based purely on the implementation, which is called white-box testing. For example an attempt can be made to ensure that every statement is executed at least once when a set of test cases is used. The acceptability of the results is determined either by resorting to the specification in addition to the implementation as a basis for test generation or by comparing the results with those produced by previous versions of the program which users have regarded as acceptable. This technique is complementary to black-box testing.

It should be noted that whereas a human tester can understand and make a certain amount of use of an imprecise document such as a specification written in natural language, an automated test generation system requires its inputs to be formal documents such as syntactically and semantically correct programs or formal specifications. We will now consider a number of strategies which have been used for the generation of tests.

RANDOM TESTING

This strategy relies solely on the domain of the input, that is the overall structure and the types of individual components of the data supplied to the program. This can be discovered from the specification, and there is no need to refer to the implementation, so this is a black-box strategy. Within the input domain, test cases are generated by selecting values at random, which can be done automatically. A possible refinement is to use a probable operational input distribution to generate the test set. It is debatable whether using values which are more likely to occur in practice increases the chances of finding

errors in the system under test, but this does help to increase the subjective level of confidence.

Random testing is intuitively the poorest strategy for selecting test cases. Early books on software testing such as Myers [1] do not discuss it except to dismiss it. A more recent experimental study by Hamlet [2] however, tends to contradict this intuitive view and suggest that there is value in random testing. It follows a report from Duran [3]. A later review by Weyuker [4] of this experimental work used a more analytical approach and is more in tune with the intuitive feeling that random testing is of little value.

Overall there are few examples in the literature of automatic test generation using this method, and there does not appear to be any general purpose system. The examples which are given are usually directed towards a specific application area such as compiler testing, where for example the Bazzichi and Spadafora system [5] generates random example programs to test Pascal compilers.

Random test case generators are easier to construct than those which use the other strategies listed below. This simplicity is attractive, but there must be doubts about how effectively a randomly chosen set of test cases will uncover errors, as from a theoretical point of view its effectiveness is still an open question (Hamlet [2], Weyuker [4], Beizer [6]), and so it might be expected that a large number of test cases will be used. This increases the importance of being able to execute the tests and evaluate the results automatically - i.e. have an oracle available to verify the results of the test. In the absence of any real consensus about the efficiency of random testing it appears that, at least in particular areas where other strategies are too complex to implement, random testing is an acceptable strategy. Specifically, it can be used to help increase confidence in the system under test as suggested by Hamlet [2].

STRUCTURAL TESTING

Using this strategy, test cases are generated by analysing the source code of the system under test. The tests are generated in such a way that different execution paths through the code are followed. For example, for each conditional statement within the code there should be at least one test case which forces each branch of the statement to be followed. For each iterative statement there should be test cases to ensure that, for example, the body of the statement is executed zero, one or many times. We only consider here pathwise generators for programs written in an imperative, sequential language as these are the most common.

There are many introductions to test generation from path coverage (e.g. Myers [1], Deutsch [7], Kopetz [8], Hetzel [9]) which differ only in the way in which paths are determined and in the criteria used to generate tests. Ideally the test cases should ensure that all possible paths for the system under test are exercised. However, in practice there may be paths through the code which cannot be followed by any test case (so-called infeasible paths), and even a restriction to those paths which are feasible would typically result in an impossibly large test set. Therefore, rather than insisting on total coverage,

weaker criteria such as execution of all statements, all branches, or all linear code sequences and jumps in the code under tests are adopted.

There is a wide range of such criteria for path coverage from which to choose, and on the basis of this choice a test case generator should examine all possible program paths and select from them. Coward [10] reviews systems which use symbolic execution to generate the paths prior to selection. The problems illustrated there include evaluation of loops, module calls, array references and path feasibility. The Casegen system of Ramamoorthy [11] tries to overcome some of these problems.

Coward gives three criteria to be satisfied before a system can be said to use symbolic execution for this purpose:

- for each path examined, a path condition should be determined.
- it should determine whether each path condition is feasible.
- for each output variable an expression should be produced in terms of input variables and constants.

A system which does not satisfy these criteria is unlikely to be suitable for automatic test generation. Six systems described in the literature satisfy Coward's criteria, namely: EFFIGY (King [12]), SELECT (Boyer [13]), ATTEST (Clarke [14]), CASEGEN (Ramamoorthy [11]), IPS (Asirelli [15])and Fortran Testbed (Hennell [16]). These systems have achieved widely varying levels of automation and we shall only describe CASEGEN here, as being typical of the technique.

CASEGEN consists of four components:

- a Fortran source code processor.
- a path generator.
- a path constraint generator.
- a constraint solving system.

The Fortran source code processor produces a flow graph, a symbol table and a representation of the source code. These are used by the path generator to produce a set of paths to cover all branches, and the path constraint generator then determines a path condition for each path. Finally, the constraint solving system generates values to satisfy each of these conditions, and these values can be used to create test cases. The strength of this system is its full coverage of Fortran but it is not yet very reliable.

Coward, trying to avoid these problems, has written SYM-BOL (Coward [17]) to make improved use of symbolic execution techniques. He has also tried to generalise path generators - which have been primarily intended for numerical software written in Fortran - to handle commercial software written in COBOL. All the common problems which have been encountered by the developers of path generators are addressed by Coward so, although SYM-BOL does not accept all of COBOL, it bodes well for future developments.

Korel [18] describes a white-box automatic test generator which is based on execution of the program under test, dynamic data flow analysis and function minimisation rather than symbolic execution. The basic steps are control flow graph construction, path selection and test generation, and these can all be automated. This project, in which the program under test must be written in a limited subset of Pascal, is at an early stage. The problems encountered when using symbolic execution have been overcome, but there are others still to be addressed. For example the system's ability to detect infeasible paths is limited. In this paper Korel suggests several possible areas for future research.

SMOTL (Bicevskis [19]) is another system of this kind. It conducts path analysis while maintaining minimum and maximum values for each variable.

Constraint-based testing (CBT) has been implemented (Demillo [20]) for Fortran-77. However this is such a new strategy that it is, as yet, difficult to evaluate it. One of the problems with this method is the difficulty of eliminating redundant or ineffective tests.

Pathwise automatic test generators are promising software development tools even if after nearly two decades of research in this field several practical problems still restrict their general use. The problems lie more in the generation of the possible paths, as illustrated by the symbolic execution approach, than with the generation of a test set which satisfies given criteria. Constraint-based testing is very promising; However for efficiency an oracle would be required. As yet the method does not cater for this.

FUNCTIONAL TESTING

Automatic test-case generators using black-box, or functional, strategies, derive the information needed to generate tests from a program's specification or test specification. They differ mainly in the type of specification used and in the sampling method. Most systems require the tester to write a specification which is oriented to testing. This specification could theoretically be used as the sole specification but this is rarely done, because of the limited expressiveness of specifications oriented towards test generation.

Function Diagram

The AGENT system (Furukawa [21]) generates test cases from a function diagram, which is an extension of a cause-effect graph [1]. A function diagram consists of a state transition diagram together with a set of Boolean functions which are defined using either a cause-effect graph or a decision table. The tests generated by AGENT satisfy natural criteria defined on the function diagram:

- they validate input and output conditions in all states.
- they exercise each transition at least once.

Other graph models such as automata (Fujiwara [22]) and Petri nets (Morasca [23]) have also been used to generate test cases using graph

coverage techniques. While all of these systems have the advantage of having well-defined criteria for selecting a test set, they all require a function diagram for the program under test. For a complex program this could be very difficult, and it seems unlikely that these systems will be widely used.

Algebraic Specifications

Jalote [24] describes the SITE system which generates test cases from the axiomatic specification of an abstract data type. The system also provides an oracle in the form of an automatically generated implementation of the specification. The test set is generated from the syntactic part of the specification by producing all possible values of the abstract data type up to a given maximum degree of complexity. Tests are then generated from the structure of these expressions.

This system is interesting because of the high degree of automation which has been achieved, but the approach has several distinctive problems: its limited scope of application; the difficulty of writing axiomatic specifications for complex data types; the lack of a theory to underpin the method of test selection; and the large number of tests generated. It is difficult to envisage the scope of this system being enlarged because axiomatic specifications are restricted in scope, and oracles generated from them are still confined to simple problems or highly restricted domains.

A method is described by Bernot [25] for constructing test cases from formal specifications and this is applied to algebraic specifications. Again, despite the case study in Dauchy [26] this approach is limited by the difficulty of writing algebraic specifications for general purpose programs. This technique offers hope for interesting developments, perhaps in the domain of model-based specification languages, because of its good foundation.

Other Approaches

Tsai [27] describes a system which generates test cases from a relational algebra query. Such queries are frequently used in database and data processing applications. The efficiency of the system is compared favourably with random testing and it seems to be efficient and reliable. Further assessment is however required. The use of relational algebra means that the system is restricted to the testing of database applications.

Another approach which has been investigated by Dyer [28] is to identify particular requirements to be tested and filter these through statistical considerations regarding operational use and risk factors. Unfortunately this method does not permit complete automation as the statistical values assigned to the requirements to be tested are largely subjective.

Recent research has thus demonstrated that automated black-box testing can be a practicable technique. The difficulties in assessing such systems lie in judging fairly the degree of automation achieved and the efficiency with which errors in the program under test are discovered. It would be a great advance if such systems were able to process established specification languages such as the notation of VDM (Jones [29]) or Z (Spivey [30]) which are already used for purposes other than testing. North [31] and Dick [32] have shown that test generation from VDM specifications is possible.

WHITE-TINTED TESTING

As indicated above, one attraction of black-box testing is that the tests are independent of the implementation to be tested. Generating tests from a model-based specification written in (for example) the notation of VDM is also appealing because it opens up the possibility of using an existing specification rather than insisting on one which has been written specifically for testing. This, therefore, is the basic approach that we have decided to follow.

Our approach however is not purely black-box and therefore differs from that of Dick, because of the impossibility of reaching certain system states using test suites. Nicholl [33] has shown that some states in model-based specifications may well be unreachable. We therefore insist that the tester have access to the source code of the system under test, in order to instantiate the system state variables with the test values generated. The exact mechanism for this is not yet decided.

It is intended that our test generation system will parse a VDM specification and record the following for each function/operation:

- the basic type of and any invariant for each variable
- the pre-condition
- the exception handling part
- the post-condition

Tests will be generated for each of these components adopting a generic strategy suitable for each kind of testing. Testing pre-conditions and the types and invariants of variables is mainly useful for front-end functions only - but this is often ignored - whereas testing the exception handling part and the post-condition is valid for every function of the specification.

The strategy adopted is, conservatively, based on partitioning of the input and/or the output space by applying a series of rules to the part of the specification considered. Constraints generation and constraints solving rules will be used, and the result of applying these rules will be *frames*. As an example of constraint generation rule the $\vee$–rule is:

$$A \wedge \neg B;\ \neg A \wedge B;\ A \wedge B;$$

– note that the last sub-domain is only generated if the $\vee$–rule is applied to a post-condition component of the specification.

Unlike Dick's system, constraints generation and constraints solving rules could be interleaved in our system, thus avoiding the explosion of sub-domains in the partition generated. Consistency checking of the expression generated will be carried out automatically. The rules will be written and applied using constraint logic programming (in Prolog III, Colmerauer [34]). The outcome of applying the generic approach will be a collection of final test frames which can be instantiated to generate tests.

In theory, this approach is complete in that it should generate all the tests usually seen as appropriate for performing black box testing. However, it is anticipated that there will be a number of limitations induced by the implementation especially at the consistency checker level where compromises will have to be made and/or manual intervention required.

Any general attempt to predict a specific result from a model-based specification must fail because such a specification may well be non-deterministic. Our system will provide an oracle by interpreting the specification using the results from the program under test, that is the part of the specification considered for the generated test will be evaluated to see whether they are satisfied regarding the test applied and the corresponding result.

Example

In his reference guide to VDM-SL Dawes [35] gives an example of a specification of a vending machine to dispense tea, coffee (with or without milk or sugar) or chocolate. The specification takes into account the price of the various drinks, available stocks of the raw ingredients and cups etc. The pre-condition for the operation GET_DRINK is

pre $(\forall i \in \text{INGREDIENTS(choice)} \cdot \text{STOCKS}(i) > 0)$
$\wedge$ (CUPS > 0)
$\wedge$ (BALANCE $\geq$ PRICES(choice)

which means that the machine has in stock all the ingredients for the chosen drink *choice*, that there is a cup in which to dispense the drink, and that the amount of money inserted, *balance*, is sufficient to pay for the drink. The original frame for this pre-condition is:

ASSUMPTIONS: $\varnothing$
CHOICE:
Drink = Tea-or-coffee | CHOCOLATE
Tea-or-coffee :: FLAVOUR: TEA | COFFEE
WHITE :|B
SWEET :|B
Ingredient = TEA | COFFEE | CHOCOLATE | MILK | SUGAR | WATER
Money = |N
Prices = Drink $\xrightarrow{m}$ Money
Stock = Ingredient $\xrightarrow{m}$ |N
inv s $\triangleq$ $\forall$ i: Ingredient $\cdot$ i $\in$ dom s
INGREDIENTS : Drink $\xrightarrow{m}$ Ingredient-set $\triangleq$

{CHOCOLATE ↦ {CHOCOLATE, WATER}} ∪
{d ↦ {d.FLAVOUR, WATER} ∪
(if d.WHITE then {MILK} else {}) ∪
(if d.SWEET then {SUGAR} else {}) | d : Tea-or-coffee}
choice : Drink
BALANCE: Money
STOCKS: Stock
CUPS: ℕ
PRICES: Prices

SPLIT:

(∃ i ∈ INGREDIENTS(choice) · STOCKS(i) ≤ 0)
∨ (CUPS ≤ 0)
∨ (BALANCE < PRICES(choice)

DATA CHOICE SET:
choice, BALANCE, STOCKS, CUPS, PRICES

When applying the rules, a series of intermediate frames is generated before obtaining a collection of final frames, one of which is:

ASSUMPTIONS:

Drink = Tea-or-coffee | CHOCOLATE
Tea-or-coffee :: FLAVOUR: TEA | COFFEE
WHITE :𝔹
SWEET :𝔹
Ingredient = TEA | COFFEE | CHOCOLATE | MILK | SUGAR | WATER
Money = ℕ
Prices = Drink $\xrightarrow{m}$ Money
Stock = Ingredient $\xrightarrow{m}$ ℕ inv s ≙ ∀ i: Ingredient · i ∈ dom s
INGREDIENTS : Drink $\xrightarrow{m}$ Ingredient-set ≙
{CHOCOLATE ↦ {CHOCOLATE, WATER}} ∪
{d ↦ {d.FLAVOUR, WATER} ∪
(if d.WHITE then {MILK} else {}) ∪
(if d.SWEET then {SUGAR} else {}) | d : TEA-or-coffee}
choice : Drink
BALANCE: Money
PRICES: Price
CUPS: $\mathbb{N}_1$
BALANCE ≥ PRICES(choice)
STOCKS(CHOCOLATE) =1
STOCKS(COFFEE) =1
STOCKS(TEA) =1
STOCKS(MILK) =1
STOCKS(WATER) =1
STOCKS(SUGAR) =0

SUGAR ∈ INGREDIENTS(choice)

CHOICE:

∅

SPLIT:

∅

DATA CHOICE:

STOCKS

The test set which will be generated for this precondition includes:

Choice = CHOCOLATE
BALANCE = 0
STOCKS = {TEA ↦ 1; COFFEE ↦ 1; CHOCOLATE ↦ 1; MILK ↦ 1;
SUGAR ↦ 1; WATER ↦ 1}
CUPS = 1
PRICES = {CHOCOLATE ↦ 1;
Tea-or-coffee(TEA, false, false) ↦ 1;
Tea-or-coffee(TEA, true, false) ↦ 1;
Tea-or-coffee(TEA, false, true) ↦ 1;
Tea-or-coffee(TEA, true, true) ↦ 1;
Tea-or-coffee(COFFEE, false, false) ↦ 1;
Tea-or-coffee(COFFEE, true, false) ↦ 1;
Tea-or-coffee(COFFEE, false, true) ↦ 1;
Tea-or-coffee(COFFEE, true, true) ↦ 1}

Choice = CHOCOLATE
BALANCE = Max_N - 1
STOCKS = {TEA ↦ 1; COFFEE ↦ 1; CHOCOLATE ↦ 1; MILK ↦ 1;
SUGAR ↦ 1; WATER ↦ 1}
CUPS = 1
PRICES = {CHOCOLATE ↦ Max_N;
Tea-or-coffee(TEA, false, false) ↦ 1;
Tea-or-coffee(TEA, true, false) ↦ 1;
Tea-or-coffee(TEA, false, true) ↦ 1;
Tea-or-coffee(TEA, true, true) ↦ 1;
Tea-or-coffee(COFFEE, false, false) ↦ 1;
Tea-or-coffee(COFFEE, true, false) ↦ 1;
Tea-or-coffee(COFFEE, false, true) ↦ 1;
Tea-or-coffee(COFFEE, true, true) ↦ 1}

Choice = CHOCOLATE
BALANCE = 0
STOCKS = {TEA ↦ 1; COFFEE ↦ 1; CHOCOLATE ↦ 1; MILK ↦ 1;
SUGAR ↦ 1; WATER ↦ 1}

CUPS = 1
PRICES = {CHOCOLATE ↦ Max_N;
Tea-or-coffee(TEA, false, false) ↦ 1;
Tea-or-coffee(TEA, true, false) ↦ 1;
Tea-or-coffee(TEA, false, true) ↦ 1;
Tea-or-coffee(TEA, true, true) ↦ 1;
Tea-or-coffee(COFFEE, false, false) ↦ 1;
Tea-or-coffee(COFFEE, true, false) ↦ 1;
Tea-or-coffee(COFFEE, false, true) ↦ 1;
Tea-or-coffee(COFFEE, true, true) ↦ 1}

CONCLUSIONS

Automatic test generators have not reached a state where it would be appropriate to make widespread use of them for testing general-purpose programs. The lack of a theoretical underpinning is partially to blame for this situation, but it should be possible to improve the tools even within the context of the current theory (Hamlet [36]).

None of the three main strategies has reached the stage where full automation has been achieved and the strategy trusted. Using functional testing - the strategy we have chosen - the standardisation of languages such as Z and the notation of VDM should provide sound bases for the implementation of testing tools. It is expected however (Bertolino [37]) that, even given further technological developments, human competence and ingenuity will remain the *sine qua non* requirement for useful testing.

ACKNOWLEDGEMENTS

The work described in this paper has been supported by SERC and by the European Commission, contract ERBCHBICT930328.

REFERENCES

1. Myers, G.J. The art of software testing. Wiley-interscience, 1979.
2. Hamlet D. and Taylor R. Partition testing does not inspire confidence. IEEE Trans. Soft. Eng. Vol. 16, No. 12, pp 1402-1411, 1990.
3. Duran J.W. and Ntafos S.C. An evaluation of random testing. IEEE Trans. Soft. Eng. Vol. 10, No. 4, pp 438-444, 1984.
4. Weyuker E.J. and Jeng B. Analyzing partition testing strategies. IEEE Trans. Soft. Eng. Vol. 17, No. 7, pp 703-711, 1991.
5. Bazzichi F. and Spadafora I. An automatic generator for compiler testing. IEEE Trans. Soft. Eng. Vol. 8, No. 4, pp 343-353, 1982.
6. Beizer B. Computing reviews, No. 9201-0020, pp. 67-67, 1992.
7. Deutsch M.S. Software verification and validation; realistic project approaches. Prentice-Hall series in software engineering, 1982.
8. Kopetz H. Software reliability. Macmillan Computer Science Series, 1979.
9. Hetzel W. The complete guide to software testing. Q.E.D. Information Sciences Inc., 1984.

10. Coward P.D. Symbolic execution systems - a review. Soft. Eng. Jnl. Vol. 3, No. 6, pp. 229-239, 1988.
11. Ramamoorthy C.V., Ho S.F. and Chen W.T. On the automated generation of program test data. IEEE Trans. Soft. Eng. Vol. 2, No. 4, pp 293-300, 1976.
12. King J.C. Symbolic execution and program testing. Comm. ACM, Vol. 19, No. 7, pp. 385-394, 1976.
13. Boyer R.S., Elpas B., and Levit K.N., SELECT - a formal system for testing and debugging programs by symbolic execution. Proc. of Int. Conf. Reliable Software, pp. 234-244, 1975.
14. Clarke L.A. A system to generate test data and symbolically execute programs. IEEE Trans. Soft. Eng. Vol. 2, No. 3, pp. 215-222 1976.
15. 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. Proc. of Fourth Int. Conf. on Software Engineering, Munich, Germany, pp. 251-263, 1979.
16. Hennell M.A., Hedley D. and Riddell I.J. The LDRA software testbeds: their roles and capabilities. Proc. of IEEE Software Fair 1983 Conference, Arlington, VA, USA, 1983.
17. Coward P.D. Symbolic execution and testing. Information and Software Technology, Vol. 33, No. 1, pp. 53-64, 1991.
18. Korel B. Automated software test data generation. IEEE Trans. Soft. Eng. Vol. 16, No. 8 , pp. 870-879, 1990.
19. Bicevskis J., Borzovs J., Straujums U., Zarins A. and Miller E.F. SMOTL - a system to construct samples for data processing program debugging. IEEE Trans. Soft. Eng. Vol. 5, No. 1, pp. 60-66, 1979.
20. Demillo R.A. and Offutt A.J. Experimental results from an automatic test case generator. ACM Trans. Soft. Eng. and Meth., Vol 2, No. 2, pp. 109-127, 1993.
21. Furukawa Z., Nogi K. and Tokunaga K. AGENT: an advanced test-case generation system for functional testing. AFIPS Press National Computer Conference, Vol. 54, pp. 525-535, 1985.
22. Fujiwara S., Bochmann G.v., Khendek F., Amalou M. and Ghedamsi A. Test selection based on finite state models. IEEE Trans. Soft. Eng. Vol. 17, No. 6, pp. 591-603, 1991.
23. Morasca S. and Pezze M. Using high-level Petri nets for testing concurrent and real-time systems. Real-time systems, theory and applications, H. Zendan (Ed.), North-Holland, pp. 119-131, 1990.
24. Jalote P. Specification and testing of abstract data types. Computer Language, Vol. 17, No. 1, pp. 75-82, 1992.
25. Bernot G., Gaudel M.C., and Marre B. Software testing based on formal specifications: a theory and a tool. Soft. Eng. Jnl. Vol. 6, No 6, pp. 387-405, 1991.
26. Dauchy P., Gaudel M.C. and Marre B. Using algebraic specifications in software testing: a case study on the software of an automatic subway. Jnl. Sys. Soft. Vol. 21, No. 3, pp. 229-244, 1993.
27. Tsai W.T., Volovik D. and Keefe T.F. Automated test case generation for programs specified by relational algebra queries. IEEE Trans. Soft. Eng. Vol. 16, No. 3, pp. 316-324, 1990.
28. Dyer M. Distribution-based statistical sampling: an approach to software functional test. Jnl. Sys. Soft. Vol. 20. No. 2, pp. 107-114, 1993.

29. Jones C.B. Systematic software development using VDM. Prentice-Hall International Series in Computer Science, 1990.
30. Spivey J.M. The Z notation, a reference manual. Prentice-Hall International Series in Computer Science, 1989.
31. North N.D. Automatic test generation for the triangle problem. NPL Report DITC 161/90, 1990.
32. Dick J. and Faivre A. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. FME '93: Industrial-Strength Formal Methods. Odense, Denmark 1993. Springer-Verlag Lecture Notes in Computer Science 670, pp. 268-284, 1993.
33. Nicholl R.A. Unreachable states in model oriented specifications. University of Western Ontario, Dept. Comp. Sc. Report No. 175, 1987.
34. Colmerauer A. An Introduction to Prolog III. Comm. ACM Vol. 33, No. 7, pp. 69-90, 1990.
35. Dawes J. The VDM-SL Reference Guide. Pitman Publishing, London, 1991.
36. Hamlet R. Special section on software testing. Comm. ACM Vol. 31, No. 6, pp. 662-667, 1988.
37. Bertolino A. An overview of automated software testing. Jnl. Sys. Soft., Vol. 15, No. 2, pp. 133-138, 1991.