Abstract

Testing is an important activity for checking the correctness of system implementations. It is performed by applying test experiments to an implementation under test, by making observations during the execution of the tests, and by subsequently assigning a verdict about the correct functioning of the implementation. The correctness criterion that is to be tested should be given in the System specifications. The specification prescribes what the system has to do and what not, and, consequently, constitutes the basis for any testing activity.

This paper describes an approach to formalization of criteria of computer systems software testing. A brief review of control-flow criteria is introduced. As a formal language for describing the criteria, the Z notation is selected. Z schemas are presented for definitions of the following criteria: statement coverage, decision coverage, condition coverage, decision/condition coverage, full predicate coverage, modified condition/decision coverage, and multiple condition coverage.

1. Introduction

Many problems in the testing process occur because Specifications are unclear, imprecise, incomplete and ambiguous. Without a specification which clearly, precisely, completely and unambiguously prescribes how a system implementation shall behave, any testing will be very difficult because it is unclear what to test. A bad system specification as starting point for the testing process usually leads to problems such as difficulties of interpretation and required clarifications of the specification's intentions. This leads to reworking of the specification during the testing phase of software development.

The use of a formal specification or model eliminates ambiguity and thus reduces the chance of errors being introduced during software development. Naturally, there still remains the issue of obtaining a formal specification that matches the actual customer requirements and this is complicated by the tendency for stated requirements to change during development. Where a formal specification exists, both the source code and the specification may be seen as formal objects that can be analyzed and manipulated. A formal specification could be analyzed in order to explore the consequences of this specification and potentially find mistakes. If this is done then we have greater confidence that we are testing the system under test (SUT) against the actual requirements. The use of a formal specification introduces the possibility of the formal and, potentially, automatic analysis of the relationship between the specification and the source code. This is often assumed to take the form of a proof, but such a proof cannot guarantee operational correctness. For this reason, even where such a proof is believed to exist, it is important to apply dynamic testing [6].

The presence of a formal specification or model makes it possible for the tester to be clearer about what it means for a system to pass a test. This may be achieved through the use of test hypotheses [2] or design for test conditions.

2. Formal methods and Testing

Software testing is mostly about empirically checking correctness. Formal methods, on the other hand, have traditionally been about the formally verifying the correctness of software. A major thrust of formal methods is the introduction of system models early in the lifecycle, against which the software can be proven through the use of appropriate mathematics. A starting point for examining the relationship between these two disciplines is to realize that both are model-based [4]. The question naturally arises, what benefit can be accrued from using testing techniques in relation to the models used in formal methods? And what benefits might there be in using the mathematical basis of formal methods in the domain of testing? By using formal methods and testing together, we can reduce the cost of development by applying testing techniques much earlier in the lifecycle while defects are relatively inexpensive to correct. We might also be able to automate more of the testing process by:

—Generating functional test cases from the specification of a system.
—Deriving provably correct oracles for checking the results of tests.

Figure 1 portrays, in the form of the V-model, the limited benefits that can be accrued to testing when the only formal model available is the code itself. Down the left-hand side each deliverable can be checked but since only the code...
is a formal object this checking is manual and depends upon interpreting potentially vague and ambiguous documents.

Fig 1. Code-based Testing

Static analysis of the code can assist testing in deriving adequacy criteria, and dynamic analysis of the code can be used to assess whether the criteria have been met. Figure 2[4] portrays the far greater possibilities that exist when formality is introduced into higher layers, the specification and design. Now, in addition to code-based testing, the following model-based test processes become possible:

—Properties of the specification can be proved. The specification can be “tested” using model checking or theorem proving.

—The specification can be validated by using testing techniques based on “executing” the abstract model through animation.

—Coverage criteria can be applied to the abstract model represented by the specification, e.g. coverage of all logical conditions, or coverage of all paths through a state chart.

This allows us to determine whether we have covered the specification as well as the code in testing [7]. Naturally, as with code-based testing, automated test generation is complicated by the feasibility problem: the problem of deciding the feasibility of a path is undecidable.

—System level functional test cases can be generated from the specification.

—An oracle for system tests can be derived for checking the results of actual test executions.

As discussed earlier, this is complicated if the types used in the specification and implementation are different, since we then need a user defined abstraction function.

—The design, and in turn the code, can be verified against the specification.

—Formal methods can suggest new kinds of execution models of code, with new kinds of test adequacy criteria.

—Properties of the code can be proved.

—Execution models can be “tested” using model checking.

—Test management can be enhanced by reasoning about, for instance, the sequencing of test cases.

The above processes that apply to the formal specification apply equally to other formalized system models, such as the design, giving a model-based approach to testing at every level of development.

3. SOFTWARE TESTING CRITERIA

Software testing criteria (or else, test data adequacy criteria or coverage criteria) play a large role of whole testing process. These criteria are used as [1]:

• stopping rules that determines whether sufficient testing has been done that it can be stopped;
• measurements of test quality when a degree of adequacy is associated with each test set;
• generators, for test data selection. Test sets are considered as equivalent if they satisfy the same criterion.

The use of testing criteria as regulatory requirements during software certification and licensing also has its own specific features and benefits. At the time of regulatory assessment, the stage of testing assessment is one of the most important where efforts of experts should be concentrated. The methods and criteria of testing are traditionally divided into structural (or white-box) and functional (or black-box) aspects. Structural testing criteria, i.e. criteria which take into account an internal structure of the program, are in turn divided into data-flow and control-flow criteria, although the combination of the two has been considered. Data-flow criteria are based on the investigation of the ways in which values are associated with variables and how these associations can affect the execution of the program. This group contains so-called all-uses, all-defs, all-p-uses and their criteria [1]. Control-flow criteria examine logical expressions, which determine the branch and loop
structure of the program. This group of criteria is considered in this paper.

3.1 Review of control-flow criteria

The following control-flow criteria are based on the well-known book by G. Myers [1]:
• Statement coverage (SC): every statement in the program has been executed at least once;
• Decision coverage (DC): every statement in the program has been executed at least once, and every decision in the program has taken all possible outcomes at least once;
• Condition coverage (CC): every statement in the program has been executed at least once, and every condition in each decision has taken all possible outcomes at least once;
• Decision/condition coverage (D/CC): every statement in the program has been executed at least once, every decision in the program has taken all possible outcomes at least once, and every condition in each decision has taken all possible outcomes at least once;
• Multiple condition coverage (MCC): every statement in the program has been executed at least once, and all possible combinations of condition outcomes in each decision have been invoked at least once.

These definitions use concepts of ‘decision’ and ‘condition’. A decision is a program point at which the control flow can divide into various paths. An example of a decision is the IF-THEN-ELSE construction in Pascal and other imperative programming languages. A decision is a Boolean expression consisting of one or several condition combined by logical connectives. A condition is an elementary Boolean expression (atomic predicate), which cannot be divided into further Boolean expressions.

All the mentioned definitions involve the statement coverage criterion as a component part. This inclusion is slightly artificial because pure decision or condition coverage is not connected with statement coverage. The purpose of this inclusion is to establish the following partial ordering of the control flow criteria: criterion \( A \) is stronger (subsumes) criterion \( B \) if every test set that satisfies \( A \) also satisfies \( B \). Other relations between testing criteria have been also considered [5].

The multiple condition coverage criterions are the strongest and require full searching of various combinations of conditions values. However, an excess of test cases can be required. If the number of conditions in a decision is equal to \( n \), then the number of test cases to satisfy this criterion is \( 2^n \); this is not normally possible in practice even for relatively moderate values of \( n \). The other mentioned criteria are weaker and require considerably less test patterns. Thus, condition coverage requires two tests for each condition and the total quantity of tests equals \( 2n \). However, in this connection, testing of combinations of conditions values is missing and such testing volume is not sufficient for safety-critical software.

An intermediate position between multiple condition coverage criterion and other criteria is taken up by modified condition/decision coverage (MC/DC) criterion:

• MC/DC: every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken on all possible outcomes at least once, and each condition has been shown to independently affect the decision’s outcome. A condition is shown to independently affect a decision’s outcome by varying just that condition while holding fixed all other possible conditions.

As evident from the definition, MC/DC criterion requires achievement of statement coverage and condition coverage criteria combined with the requirement that each condition should affect the decision’s outcome. It means that the outcome of a decision changes as a result of changing a single condition. This criterion requires testing of various (but not all) combinations of conditions values but the number of tests still grows by linear law.

Full predicate coverage criterion [3] is similar to MC/DC but is weaker. This criterion is based on the specifications and the original definition uses different terms (for example, ‘clause’ instead ‘condition’). But it is possible to reformulate it for uniformity with previously defined criteria and using a definition from as a base:

• Full predicate coverage (FPC): every statement in the program has been executed at least once, and each condition in a decision has taken all possible outcomes where the value of a decision is directly correlated with the value of a condition. This means, a decision changes when a condition changes.

Thereby, the distinction between full predicate coverage and MC/DC is that while varying a condition, it is not necessary to fix all other possible conditions. In this case it is easier to assemble test cases than for MC/DC. But on the other hand, the effect of varying a condition on a decision can be masked by the other conditions.

4. Formalization of control-flow criteria in Z

The criteria such as full predicate coverage and modified condition/decision coverage are quite complicated and require additional explanation. Various understandings of these definitions are possible giving rise to ambiguity. To help alleviate this situation, the elaboration of formalized definitions of testing criteria, which describe criteria contents in rigorous mathematical form, is presented here.

Formalization of certain criteria has been carried out using set theory, graph theory, predicate logic, temporal logic. In this paper, the Z notation [5] is used for the formal definition of the criteria.

The reasons for choosing the Z notation are follows. Z has been used for a number of digital systems in a variety of ways to improve the specification of computer-based
systems. Many textbooks on Z are now available. The teaching of Z has become of increasing interest. Concerning software testing, the Z notation has been used to derive tests from model-based specifications, for the testing of abstract data types (modules, classes, package, clusters), automatic test case generation [2], the selection of test cases and evaluation of test results. So, when Z is used for software development and testing, it is expedient to use testing criteria, also formulated in Z.

4.1 Basic concepts

The following two given sets are used further for definitions of testing criteria:

\[
\text{[INPUT, STATEMENT]}
\]

The first set is a set of possible values of input program variables. So, any \( i \in \text{INPUT} \) is a vector whose components are specific values of all input quantities. The second set is a set of all program statements. The further specification of the nature of these sets is not necessary for the creation of the criteria definitions.

When the specific values of input variables are determined and the program is executed, usually only a part of program branches are involved, i.e. only a part of program statements are activated. Below \( \text{path } i \) is a set of program statements that are executed when the input variables value is \( i \).

\[
\text{| path: INPUT} \rightarrow \text{P STATEMENT}
\]

The set \( \text{Bool} \), used for the determination of the value of conditions, contains only two elements – 0 and 1.

\[
\text{Bool} == \{0, 1\}
\]

\( \text{PartInput} \) is a set of all non-empty subsets of \( \text{INPUT} \), excluding the set \( \text{INPUT} \).

\[
\text{PartInput} == \text{P1 INPUT} \setminus \{ \text{INPUT} \}
\]

The abbreviation \( \text{cond} \) is introduced for the set of all possible conditions, which are considered as logical predicates on \( \text{INPUT} \), i.e. for any \( i \in \text{INPUT} \) the value of a condition equals 0 (FALSE) or 1 (TRUE). The set of conditions will later be restricted and conditions linked with each specific decision will be used.

\[
\text{Cond} == \text{INPUT} \rightarrow \text{Bool}
\]

The following schema1 describes the notion of ‘decision’. On the one hand, each decision is a statement of a program (variable \( \text{decst} \)). On the other hand, each decision is a logical predicate (the function value) on the subset \( \text{decinput} \) of input data, for which the executed path passes through this decision, i.e. \( \text{decst} \in \text{path } i \). Correspondingly, \( \text{decinput}0 \) and \( \text{decinput}1 \) are the sets, where the function value (i.e., the value of the decision) equals 0 and 1 respectively. These sets partition the set \( \text{decinput} \), i.e. their union equals \( \text{decinput} \), and their intersection is the empty set. From the definition of \( \text{PartInput} \), both of them are nonempty. Thereby, the decisions, which are either all 0 or all 1, are excluded from consideration. The reasons are that all such decisions are covered when the statement coverage criterion is satisfied and that ‘affect the decisions outcome’ (see above the definition of the MC/DC) and ‘the correlation with the value of a condition’ (see above the definition of the full predicate coverage) are impossible for such degenerate decisions.

\[
\begin{align*}
\text{dec} & \quad \text{decst : STATEMENT} \\
\text{decinput} & \quad \text{P1 INPUT} \\
\text{decinput}0, \text{decinput}1 & \quad \text{PartInput} \\
\text{argdec} & \quad \text{P1 cond} \\
\text{value} & \quad \text{cond}
\end{align*}
\]

\[
\begin{align*}
\text{decinput} & == \{i : \text{INPUT} | \text{decst} \in \text{path } i\} \\
\langle \text{decinput}0, \text{decinput}1 \rangle & \quad \text{partitions } \text{decinput} \\
\text{argdec} & == \{c : \text{cond} | \text{dom } c = \text{decinput} \land \text{ran}(\text{decinput} < c) = \text{Bool}\} \\
\text{dom } \text{value} & = \text{decinput} \\
\text{decinput}1 & == \{i : \text{INPUT} | \text{value } i = 1\}
\end{align*}
\]

The set \( \text{argdec} \) contains all conditions that are components of a given decision. Each condition should take both 0 and 1. This means that conditions that are either all 0 or all 1 are not considered. The reasons are the same as for decisions – each such condition is always covered when a decision is covered and varying condition’s value (for MC/DC and full predicate coverage) is impossible for such degenerate conditions. To describe the process of testing, the set of testing data is introduced. This set, named test set, can satisfy or not satisfy to testing criteria.

4.2 Formal definitions

The following schema gives the definition when testing data satisfy the statement coverage criterion.
As the statement coverage criterion is a component part of all other criteria, its schema is in the signature of all other schemas used for defining testing criteria. Thus, the set testset is used (via the StatementCoverage schema) for definitions all testing criteria.

The following schema determines a formal definition of the decision coverage criterion based on the fact that for any specific decision \( d \) the set testset should overlap with both \( d.decinput0 \) and \( d.decinput1 \). In this case, the given decision takes both the value 0 (for the testing data from \( d.decinput0 \)) and the value 1 (for the testing data from \( d.decinput1 \)).

The following schema determines the condition coverage criterion and is analogous with the previous schema. It claims that a pair of input data from testing set should exist, for which the condition takes different values (both 0 and 1).

The formal description of the decision/condition coverage criterion uses the fact that this criterion is the union of the decision criterion and the condition criterion. So, the schema of this criterion contains only references to two previous schemas.

The requirement, which determines the full predicate coverage criterion, is similar to the corresponding requirements for the condition coverage: a testing set should contain such input data \((i0\) and \(i1)\) that the value of a condition equals 0 for one of them and equals 1 for the other.

But if in case of condition coverage there are no restrictions on the input data (except their membership in \( d.argdec \); i.e., or these input data the decision, as a program statement, should be executed), then for full predicate coverage there is the additional restriction: for these input data the decision also should have different values, i.e., both the condition and the decision should vary simultaneously.

The following schema determines the MC/DC criterion. As the MC/DC criterion subsumes the full predicate criterion, this schema contains the FullPredicateCoverage schema, combined with an additional restriction. This restriction requires that, in case varying of one condition, all other conditions \((othercond)\), which make up the given decision, should not vary, if it is possible in principle. It means that the values of each \( othercond \) for \( i0 \) and \( i1 \) are equal.
The subset of pairs of input data, which display this combination, is denoted as pair. If such a non-empty subset pair exists, then at least one from its elements should be a member of the testing set. This formal definition eliminates a certain shortcoming of the definition in natural language. The original definition does not clearly describe the situation when, varying one condition, it is impossible to fix all other conditions. It is not necessarily the best option to consider that the MC/DC is not satisfied in this situation. The formal definition shows that testing data should answer the full predicate coverage criterion for this condition. It means that it is acceptable to vary this condition simultaneously with the decision but without fixing the value of all other conditions.

The last scheme determines the multiple condition coverage criterion.

\[
\forall d : dec; \condset : \text{P cond} \mid \\
\condset \in \text{P d.argdec} \bullet \\
( \exists \comb : \text{P1 d.decinput} \bullet ( \forall i : \comb \bullet ) \land \\
( \forall c : \condset \bullet c i = 1 ) \land \\
( c i 0 \neq c i 1 ) \land \\
( \forall c : d.argdec \bullet c \notin \condset \bullet c i = 0)) \Rightarrow \\
( \text{testset} \cap \comb \neq \emptyset )
\]

The definition claims that the testing data set (testset) should contain the data for testing every combination of the values of conditions into a decision, if such combination is possible in principle (i.e., if there are input data for which the value of conditions make up the given combination). In the schema given above, each combination of the values of the conditions is clearly defined by the subset condset of the conditions, which equal 1 for this combination. Accordingly, the other conditions from d.argdec, which are not members of condset, equal 0 for this combination. The subset of input data, which display this combination, is denoted as comb. If such a non-empty subset comb exists, then at least one from its elements should be a member of the testing set.

5. Conclusion

The subject of this paper is the formalization of criteria for complex computer systems software testing. Control flow criteria, i.e., criteria using logical expressions, which determine the branch and loop structure of the program, are considered. This group includes well-known criteria [1] and relatively new criteria – full predicate and modified condition/decision coverage (MC/DC) criteria. A brief review and examples illustrating the testing criteria are introduced. The Z notation is used for the formal definition of the criteria. Z schemas formally describing all main control flow testing criteria are presented.

This paper has reviewed the state of the art regarding ways in which formal specifications can be used to assist in software testing. In doing so, it has raised a number of issues and we have seen that there have been differing degrees of success with these issues across the formalisms.

The proposed approach could also be used for the formalization of other testing criteria (e.g., data-flow control criteria). Another direction for future work could be using formal definitions for detailed analysis of the content and applicability of the most complicated existing criteria and for producing new criteria. One of the possible criteria for further analysis is the modified condition/decision coverage criterion.

6. REFERENCES