ISO/IEC/IEEE 29119-4:2015 Software and Systems Engineering - Software Testing - Part 4: Test Techniques

João Brito, M9984
Department of Computer Engineering
University of Beira Interior
Covilhã, Portugal
joao.pedro.brito@ubi.pt

Tomás Jerónimo, M9988
Department of Computer Engineering
University of Beira Interior
Covilhã, Portugal
tomas.jeronimo@ubi.pt

***Abstract*—This document aims at going through the ISO/IEC/IEEE 29119-4:2015 standard, highlighting the most relevant aspects, while attempting to take away some of the formal nature and replacing it with a lighter tone.**

I. Introduction

The ISO/IEC/IEEE 29119-4:2015 standard provides guidelines on the methodology and implementation of software design techniques. When applied, they can help improve the efficiency and accuracy of any given piece of software. Moreover, precise steps are described, along with several distinct techniques, to help achieve the desired system, one that fails no test and stands firm before any situation.

II. Background and Foundational Concepts

In this section, we provide some thought-provoking questions accompanied by important concepts and definitions.

A. What is the need for a standard?

Without diving deeply into this topic, one might argue that there is no need for testing standardization. There is a solid chance that a simple test can give the developer the answer to the dreaded question "Does this program work?", to which can follow "If a simple test is enough, why consider more robust methods?". Put simply, the former can be deceiving and the latter loses meaning when the benefits of standardized techniques reach full form.

A well established list of guidelines has the power to surpass the capabilities of more basic methodologies, speed up the development pace and ease the inevitable frustration brought forward by the debugging process.

B. Concepts and Definitions

In the current context, a *test item* is any kind of system, software item, class, object, document or specification that is being tested. In each of the techniques that we will go over in this paper, three activities play an equally important role: *test conditions*, *test coverage items* and *test cases*. These need to be determined as accurately as possible, keeping in mind the technique used.

A *test condition* is a testable aspect of a *test item*, such as a function, transaction, feature, quality attribute or structural element identified as the basis for testing. Being the first activity to be carried through, it is vital to decide which attributes are to be tested and which are to be left out.

The next task involves the specification of *test coverage items*, characterized by the attributes of a *test condition* that can be covered during testing. Note that a single *test condition* can be the basis for one or more *test coverage items*.

Finally, a *test case* is a set of predictions, inputs (including actions, where applicable) and expected results, developed to determine whether or not the covered part of the *test item* has been correctly implemented.

III. Specification-based Testing Techniques

The following section will be dedicated to presenting the techniques under the definition of specification-based testing. This collection of techniques is often called *Black-Box Testing* due to the inherently invisible nature of the *test items* evaluated. In other words, the internal structure (e.g. source code) of said items is not known or visible.

A. Equivalence Partitioning

This well known technique tries to divide the input and output into subsets (called "equivalence partitions") where all the values contained are similar in a certain way. It can prove to be very time saving, especially with large value domains.

The ISO/IEC/IEEE 29119-4:2015 standard recommends:

1) *Test conditions:* every equivalence partition;
2) *Test coverage items:* every equivalence partition;
3) *Test cases* (two approaches):
   a) one-to-one, a *test case* for every partition; OR
   b) minimized, the least amount of *test cases* to cover all partitions.

Chose an approach and select *test coverage items* along with real input values; compare the output to the expected results; repeat these steps until the desired level of test coverage is achieved. Every technique shown onwards uses a similar strategy for creating *test cases*.

For example, if we were to see if an integer (e.g. 20) fits within a certain interval of other integers (e.g. 0 and 100), we could create *test cases* to go over every possible integer from 0 to 100 and check our results. But there is a smarter way to go about doing this. Take an arbitrary number between 0 and 100 and see if the program says that it does belong to that interval. After all, in this situation a number like 20 is the same as, say, 44. Add another case where the input is smaller or bigger than 0 or 100, respectively. With much less *test cases*, the program was verified.

B. Classification Tree Method

This technique uses a model resembling a tree, thus forming a disjoint set of classes (or classifications), which can be further split into subsets. The input domain forms the root node, followed by the classifications (that shall be the *test conditions*) and leaving the sub-classes as leaf nodes. It's main advantage lies with the visual representation of test-relevant aspects (aspects that affect the behaviour of the test object during the test).

The *test coverage items* shall be derived in one of two ways: "minimized", where classes are included in *test coverage items* such that the minimum number of these is used to cover all classes OR "maximized", in which there is, at least, one *test coverage item* for every class.

C. Boundary Value Analysis

The following technique serves as a compliment to Equivalence Partitioning (III-A), given that it encourages testing on the boundaries, i.e. the limits of the values used. Boundaries are sometimes overlooked, so this type of testing focuses on them (each boundary is a *test condition*). The *test coverage items* can either be comprised of the boundary value + a value outside the boundary by a given distance, called *step* (e.g. *boundary value* = x and *value outside* = x + *step*) OR the former plus a value inside the boundary (e.g. *value inside* = x - *step*, *boundary value* = x and *value outside* = x + *step*).

Though seemingly easy, it's important to apply this method correctly, especially when it is so common to forget edge cases (like the boundaries mentioned).

D. Syntax Testing

Every *test item* requires input in a specific format, so this tool tries to feed a *test item* with valid, as well as, invalid inputs. It's worth nothing that if the software we are testing is so catered to a specific type of input (e.g. an application with a text field for numbers and the keyboard only has numbers), this technique does little to nothing in the way of improving the tests. Nonetheless, there are many cases susceptible to failure from naively or intentionally wrong inputs.

In this case, the *test conditions* are either the entire input domain or a part of it. As for the *test coverage items*, they can either be positive (namely, "options" of the defined syntax, valid inputs) or negative (invalid mutations of said syntax).

The ISO/IEC/IEEE 29119-4:2015 standard recommends the use of as many *test cases* as possible, namely (but not limited to):

- whenever an option is presented, exploit that and add a *test case* for every option;
- whenever a certain quantity of letters or numbers is needed, add a *test case* with fewer than required characters and another with more;
- whenever a limited number of valid choices is allowed, add *test cases* with inputs that were not specified.

E. Combinatorial Test Techniques

When we are asked for more than one input, there are several ways to check a *test item's* resilience. Despite fundamental differences, the following variations consider a *test condition* to be a chosen test item parameter (P) taking on a specific value (V), resulting in a P-V pair:

- All Combinations, where every possible combination of inputs is tested. An exhaustive method like this only makes sense with small, feasible input domains;
- Pair-wise, a smarter version of the previous method that tries to include, within a *test case*, several tests at once. Instead of testing just one parameter and locking the others, (if possible) shuffle them as well to test as many as possible in one *test case*;
- Each Choice, a strategy where every value of every parameter must be used at least once. This does not cover the entire domain of *test cases*, but gives a relatively decent approximation;
- Base Choice, where we define a base choice, i.e. a combination of values that we are going to keep semi-fixed, as the remaining get shuffled. As an example, let's consider 3 variables (e.g. a, b, c forming the triple (a,b,c)), their possible values (e.g. a = {1,2,3}, b = {2,3,4}, c = {3,4,5}) and a base choice (e.g. a = 1, b = 2, c = 3). Now, at a time, one value of the base choice remains fixed while the others change, e.g. fixing a and c' values we get (1,2,3), (1,3,3) and (1,4,3). After applying the same mechanic to the fixed pairs a&b and b&c, we get the final *test cases*.

F. Decision Table Testing

This simple technique is useful for documentation an visualization. Consists in a table where the possible conditions are combined, with the logically corresponding actions derived from them (both the *test conditions* and *test coverage items* are composed of the aforementioned conditions and actions). As an example:

		Test case 1	Test case 2	Test case 3
conditions	color	red	blue	green
	size	small	small	big
	type	car	bike	car
actions	parking	✓	✓	-

G. Cause-Effect Graphing

Sometimes confused with the previous technique, this method takes the conditions (or causes) and combines them to reach a logical conclusion (or effect). It's visually represented

as a graph with cause and effect nodes. If required, the graph can be converted into a decision table.

Just like the previous method, the *test conditions* and *test coverage items* are formed by the causes-effects pair.

H. State Transition Testing

Another visual technique, consists in representing every state a *test item* can be in as a graph. There is usually a sense of direction, with start nodes close to each other and, as we move along, another cluster of end nodes. To create *test cases*, it is as simple as choosing one of the possible paths from start to finish.

Recalling the formal terminology, the *test conditions* can be the entire state model, all transitions or all states. Similarly dependant on the implementation choice, *test coverage items* can consist of a given number of transitions (valid and invalid) or the states themselves.

I. Scenario Testing

This technique uses a model of sequences of interactions that play out between the *test item* and other systems (including the user). Therefore, the *test conditions* and *test coverage items* shall be a sequence of interactions. When establishing this method, it is important to identify the "main" scenario, i.e. the expected sequence of actions performed by the *test item*, and the "alternative" scenarios, i.e. non-main sequences that may or may not be taken by the *test item*.

J. Random Testing

The final "black-box" technique covered in the ISO/IEC/IEEE 29119-4:2015 standard attempts to automate the testing process by taking the entire input domain and, using an input distribution (e.g. normal distribution), generating random *test cases* in a loop of automated testing. The present standard also considers the *test conditions* to be the entire input domain (there are no recognised *test coverage items* for Random Testing).

It's main difficulties lie with the development of a generator and an "oracle", which tells if the *test item* produced the right output or not. If the output is correct, nothing extra is done. On the contrary, if the output fails to meet expectations, the *test case* is saved for further analysis.

To summarize, this method is especially useful to save time and human resources, given that the *test cases* are computer generated, rather than handmade.

IV. Structure-based Testing Techniques

The testing techniques presented in this section serve two main purposes:

- test coverage measurement;
- structural *test case* design.

These techniques can also be called *White-Box Testing*, as they are used to test the internal structure of the *test item*. First, they are used to assess the amount of testing performed by tests derived from specification-based techniques, to assess their coverage. Then they are used to design additional tests in order to increase the test coverage.

Before introducing these techniques, it is important to introduce the concept of *test coverage*. *Test coverage* measures the amount of testing performed by a set of test techniques. This measurement is done by calculating the percentage of *coverage items* that have been successfully exercised, which can be done by using the following formula:

$$Coverage = \frac{Nº\,of\,coverage\,items\,executed}{Total\,number\,of\,coverage\,items} \times 100\% \quad (1)$$

There is a drawback in using coverage measurement as it can check the flow of different paths in the program but it can not, for example, test the false condition in it. This means that two different *test cases* may achieve exactly the same coverage but the input data of one may find an error that the input data of the other does not.

A. Statement Testing

This technique is used to measure the number of statements in the source code which can be executed given the input data.

Each executable statement shall be considered a *test condition*.

The tester must then choose a sequence of executable statements and the test inputs that will cause this sequence to be exercised. The test inputs must then be applied and the coverage percentage calculated. If the test inputs don't produce the required coverage new test inputs must be derived.

B. Branch Testing

This technique aims to cover all the branches in the control flow of the *test item*. To do so, every branch in the control flow of the *test item* must be identified and shall be a *test condition*. In the ISO/IEC/IEEE 29119-4:2015 a branch is defined as:

- a conditional transfer of control from any node in the control flow model to any other node; OR
- an explicit unconditional transfer of control from any node to any other node in the control flow model; OR
- when a *test item* has more than one entry point, a transfer of control to an entry point of the *test item*.

During the *test case* derivation if there are sequences of executable statements that reach one or more *test coverage items* that have not yet been executed, then new test inputs must be determined and applied. This process must be repeated until the results provide the required level of test coverage.

C. Decision Testing

This technique aims to cover each decision outcome in the *test item*. The ISO/IEC/IEEE 29119-4:2015 defines decisions as points in the *test item* where two or more possible outcomes may be taken by the control flow. It is also stated that decisions can be used:

- for simple selections;
- to decide when to exit loops;
- in case statements.

This testing technique allows for better coverage than the *Statement Testing* (IV-A) technique as it requires more checks, and a perfect decision coverage always guarantees a perfect statement coverage.

D. Branch Condition Testing

This testing method's objective is to ensure that each one of the possible branches from each decision, within the *test item*, is executed at least once and thereby ensuring that all reachable code is executed. This helps in validating that no branch leads to abnormal behavior of the *test item*. In this technique every Boolean value of the logical conditions must be identified as a *test coverage item*.

It is important to mention that even though this method is stronger than *Decision Testing* (IV-C), it is not possible to cover all the paths as it can be seen in the following example:

```
if (a && b) {expression 1}
if (c || d) {expression 2}
```

If the following values for the conditions *a* and *b* are selected:

```
a = true, b = false
a = false, b = true
```

A 100% coverage is still achieved but the *expression 1* is never met.

E. Branch Condition Combination Testing

As it was mentioned in the previous section, *Branch Condition* (IV-D) is not able to cover every path of the *test item*. To cover every path it is necessary to execute all possible combinations of conditions and this is what this approach allows. This means that each unique feasible combination of Boolean values must be identified as a *test coverage item*.

F. Modified Condition Decision Condition (MCDC) Testing

This technique is the middle ground between *Branch Condition* (IV-D) and *Branch Condition Combination* (IV-E). Just like the previous methods, in MCDC testing each condition must be a *test condition* but each condition should also affect the decision's outcome independently. This means, if we were to test a condition, we should not use the values for other conditions that mask this condition results. For example:

```
if (a && b) {expression 1}
```

If *a* = *false*, the decision outcome will always be *false*, so it will mask results for *b*.

G. Data Flow Testing

Data Flow testing is a testing technique that focuses on the variables and their values used within the *test item*. This testing method keeps check of the points at which variables receive values and the points at which these values are used. The idea is to reveal the coding errors and mistakes, which can result in improper implementation and usage of the data variables or data values in the programming code.

Categories are assigned to variable occurrences where the category identifies the definition or the use of the variable at that point.

According to the ISO/IEC/IEEE 29119-4:2015, "definitions" are variable occurrences where a variable is possibly given a new value, and a "use" is an occurrence where the variable is not given a new value. "Uses" can be further distinguished as either "p-uses" (predicate-use) or "c-uses" (computation-use). A p-use is when the variable is used to determine the outcome of a condition such as a while-loop or if- then- else. Conversely, c-uses are all others occurrences including when a variable is used as an input for the definition of any variable, or of an output.

In data flow testing, each definition-use pair for a variable in the *test item* is a *test condition*.

This testing technique is used to identify any of the following issues:

- a variable that is declared but never used within the program;
- a variable that is used but never declared;
- a variable that is defined multiple times before it is used;
- deallocating a variable before it is used.

However, currently all of this potential issues are efficiently detected by compilers, thus making such verifications not worth the effort (for the most part).

V. Experience-based Testing Techniques

This section presents a method of executing testing activities with the help of experience gained in the past. The tester verifies and validates software quality using his past experience of testing similar type products. This type of testing is required when there is:

- limited knowledge of the software product;
- inadequate specification;
- restricted amount of time to perform testing.

A. Error Guessing

This technique involves the draft of a checklist of defects that may exist in the *test item*. The tester must then identify inputs to test the *test item* that may cause failures, if those defects do exist. Each defect is considered a *test condition*. This is the least formal technique as there are no particular rules to design the *test conditions*.

VI. Conclusions

The aim of this document was to present a number of test techniques, presented in the ISO/IEC/IEEE 29119-4:2015 standard, to aid the software development process. The three main sections of this paper showcased a wide variety of methods that testers can choose from, depending on their needs and limitations.

Whenever the tester can not access the source code, the techniques listed in section III allow tinkering with the input parameters in order to validate the expected outputs. In the following section (IV), it is implied that the source code is accessible, which allows the amount of testing performed by

a set of test techniques to be measured. The last section (V) resorts to the tester's past experience to validate the quality of the software.

Even though these guidelines are competent in their domain, they should always be coupled with the best industry practices.

REFERENCES

[1] ISO 29119-4:2015 (2015). Software and systems engineering - Software testing - Part 4: Test techniques. Standard, International Organization for Standardization, Geneva, CH

[2] British Computer Society Specialist Interest Group in Software Testing (BCS SIGIST). Standard for Software Component Testing (2001). [Online] http://www.testingstandards.co.uk/Component%20Testing.pdf

[3] SysGears. Test Design Techniques overview, 2016. [Online] https://sysgears.com/articles/test-design-techniques-overview/equivalence-classes

[4] Guru99. Code Coverage Tutorial: Branch, Statement, Decision, FSM, 2019. [Online] https://www.guru99.com/code-coverage.html