

Complexity of Equivalence Class and Boundary Value Testing Methods

Vineta Arnicane

University of Latvia, Raiņa Blvd 19, Rīga, Latvia
vineta.arnicane@lu.lv

There are two groups of domain testing methods – equivalence class testing (ECT) methods and boundary value testing (BVT) methods reviewed in this paper. This paper surveys 17 domain testing methods applicable to domains of independent input parameters of a program. This survey describes the basic algorithms used by domain testing methods for test case generation. The paper focuses on the theoretical bounds of the size of test suites or the complexity of domain testing methods. This paper also includes a subsumption hierarchy that attempts to relate various coverage criteria associated with the identified domain testing methods.

Keywords: software testing, domain testing, equivalence class testing, boundary value testing, equivalence partitioning, partition analysis, boundary value analysis.

1 Introduction

A domain of a program with mutually independent parameters is a set of all combinations of all values of these parameters. The input domain can be very big. The main goal of domain testing methods is to achieve a test suite the size of which is considerably smaller than the count of all inputs of the program, and which effectively reveals failures of the program as much as possible.

The main approach is to divide the test object's input domain in subdomains or equivalence classes so that inputs from the same subdomain are processed in the same way, that is, cease the same type of program's output or behavior [1, 2]. Any input represents all inputs of the subdomain to which it belongs. Assumption of the approach is that if the few good representatives of the class are tested, most or all of the bugs that could be found by testing every member of the class are found [1–5], and if representatives do not catch the bug, the other elements of class will not either [3–5].

The equivalence classes of the program's input domain can be obtained using both the program's structural analysis – path analysis, and functional analysis – equivalence partitioning, domain analysis, boundary value analysis [3].

Path analysis approach is known since the 70s of the 20th century. It assumes that domain error occurs if an input traverses the wrong path through a program [6]. Symbolic

execution can be used to obtain path constraints [7] or system of inequalities [6, 8–10], solution of which determines the subdomain of input domain, ensuring execution of this path.

Partition analysis technique has developed since the 80s of the 20th century [11–15]. Equivalence partitioning technique subdivides input domain in equivalence classes on the basis of the program's specification [1–5]. The first step is to determine the domain of valid values for each of the program's parameters. These are the equivalence classes of all valid input values, one class for each parameter. Second step is to establish the domain of all invalid input values for each parameter. Invalid values are values that are possible for the input parameter but are not included into the class of valid values. By subsequent steps, the equivalence classes are refined in such a way that each different requirement and each different program behavior is a response to an equivalence class.

The same principle of dividing into equivalence classes can be used for output data, too. As a result, the equivalence classes of input values are refined according to the principle that if two different inputs of program cause outputs that belong to different equivalence classes of outputs, these inputs should be in different equivalence classes of input values, too [1, 16].

Boundary value analysis devotes special attention to boundaries of equivalence classes, because praxis shows that boundary values often reveal faults. Boundary value testing methods for test cases choose boundary values, special values, as well as values that are close to them – just above them or just below them.

There is the group of methods that different authors call domain testing methods or domain analysis methods [17]. These methods also take into account dependencies or interactions between input parameters [3, 4, 17–20]. By these methods, the input domain often is seen as a geometrical shape and its edges – as boundaries. In most cases the domains with linear boundaries can be examined [3, 17–21], but there are some methods that allow to test nonlinear boundaries, too [4, 21–23].

Using path analysis and equivalence partitioning, equivalence classes are determined. From each class one value can be chosen as a representative of the class for testing. When we have some parameters of the program with some equivalence classes for each of them, we should use a strategy how to combine them in test cases. A similar situation is with values obtained by boundary value analysis. These approaches to domain testing methods in this paper are accordingly called equivalence class testing (ECT) methods and boundary value testing (BVT) methods.

The complexity of domain testing methods is the smallest size of the test suite generated by a method. There are two parameters for comparing the testing methods – relative effectiveness and cost [25]. The complexity of the method is closely related with the cost of the method. If complexity is high, it means that there might be a large count of test cases in the test suite. As the size of the test suite grows, more resources are necessary for testing, for instance, testers, hardware, software, time, budget. Secondly, we should consider the effectiveness of the method. A method is considered effective if the software tested thoroughly according to that method is almost correct [25]. But method should be efficient too – if the test case fails, it is better if there is a small count of candidates (input values of the test case) to blame.

Hence, it is very important to take into account the method's complexity and effectiveness during the planning stages of testing.

The aim of this paper is to review some ECT and BVT methods mentioned in literature and assess their complexity. The paper focuses on the theoretical bounds of the size of test suites or the complexity of domain testing methods. This paper also includes a subsumption hierarchy of the identified domain testing methods.

Section 2 gives some definitions useful in the context of this paper, section 3 explains the testing criteria of domain testing methods. There is a review of equivalence class testing methods in section 4 and of boundary value testing methods in section 5. In sections 4 and 5, the algorithm of testcase generation is shortly described and the complexity of each identified domain testing method is assessed. Section 6 summarizes the complexity of domain testing methods and provides the hierarchy of domain testing methods according to subsume ordering. Finally, section 7 concludes this survey with a summary of most important results and some future directions of research.

2 Definitions

Suppose that P is a program with N input parameters X_i , where $1 \leq i \leq N$. For each parameter, input domain D_i is partitioned into M_i equivalence classes with extreme points as boundary values.

The meaning of boundary value testing is to examine the program when input parameters assume extreme values for each equivalence class (maximal, minimal), just above (for some small value ϵ) or just below the extreme values, and when value is *nominal* – inside the equivalence class in distance from extreme values that is considerably bigger than ϵ .

For the corresponding domain D_i of each parameter's X_i , each equivalence class d_{ij} of the ordered elements can be graphically represented as showed in Fig. 1. The minimal boundary value of class is $x_{ij \min}$, maximal boundary value is $x_{ij \max}$, where $1 \leq j \leq M_i$. Nominal value of the class is $x_{ij \text{nom}}$. Values $x_{ij \min-}$, $x_{ij \max-}$ are a little smaller than appropriate boundary values, but $x_{ij \min+}$, $x_{ij \max+}$ are a little bit bigger. For the sake of simplicity, *min-*, *min*, *min+*, *nom*, *max-*, *max*, *max+* instead of $x_{ij \min-}$, $x_{ij \min}$, $x_{ij \min+}$, $x_{ij \text{nom}}$, $x_{ij \max-}$, $x_{ij \max}$, $x_{ij \max+}$ will be used in this paper in cases when it cannot cause misunderstanding.

Boundary values $x_{ij \min}$ and $x_{ij \max}$ may belong to an equivalence class, but they can also be excluded from it. Nevertheless, they are boundary values for this class.

The following inequalities hold for each i, j , when $1 \leq i \leq N$ and $1 \leq j \leq M_i$.

$$\begin{array}{lll} x_{ij \min} - x_{ij \min-} \leq \epsilon_{ij} & x_{ij \min+} - x_{ij \min} \leq \epsilon_{ij} & x_{ij \text{nom}} - x_{ij \max} \geq \epsilon_{ij} \\ x_{ij \max+} - x_{ij \max} \leq \epsilon_{ij} & x_{ij \max} - x_{ij \max-} \leq \epsilon_{ij} & x_{ij \max} - x_{ij \text{nom}} \geq \epsilon_{ij} \end{array}$$

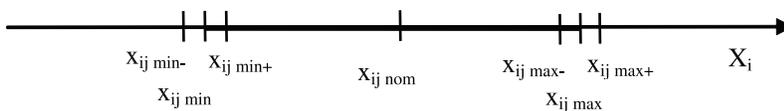


Fig. 1. Equivalence class d_{ij} : $[x_{ij \min}, x_{ij \max}]$, its boundary values $x_{ij \min}$, $x_{ij \max}$, inner OFF points $x_{ij \min+}$, $x_{ij \max-}$, outer OFF points $x_{ij \min-}$, $x_{ij \max+}$ and nominal value $x_{ij \text{nom}}$

Let us call values just above the minimal value and just below the maximal value *inner OFF* points (they are inside equivalence class) and values just below the minimal value and just above the maximal value *outer OFF* points.

We have two assumptions:

- 1) For each parameter X_i , conjunction of all equivalence classes is domain D_i .

$$\text{So, } D_i = \bigcup_{j=1}^{M_i} d_{ij} \text{ for } \forall i, j,$$

where $1 \leq i \leq N$ and $1 \leq j \leq M_i$.

- 2) There are no common values between equivalence classes – $\forall i, j, k$,

$$\text{where } 1 \leq i \leq N, 1 \leq j \leq M_i, 1 \leq k \leq M_i \text{ and } j \neq k \text{ } d_{ij} \cap d_{ik} = \emptyset.$$

Although for each parameter, equivalence classes do not have common values, they can have common boundary values. For instance, as it is shown in Fig. 2, if domain for X_i consists of a closed interval $[x_{ia}, x_{ib}]$ and left-open, right-closed interval $(x_{ib}, x_{ic}]$, they have a common boundary value x_{ib} . Let us denote the size of the set of all common boundary values for parameter X_i with L_i .

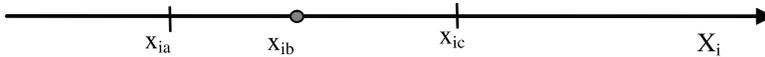


Fig. 2. Common boundary value x_{ib} for intervals $[x_{ia}, x_{ib}]$ and $(x_{ib}, x_{ic}]$, $L_i = \{x_{ib}\}$.

For each parameter X_i , values which belong to its domain D_i are called *valid* values, all other values – *invalid*. Invalid values can also be divided into equivalence classes. The size of the set of equivalence classes of invalid values for parameter X_i in this paper is denoted by Q_i .

Boundary value testing methods are also applicable in cases when the parameter’s domain is a set of discrete elements. It is important in such cases that there exists any function according to which elements of the domain can be ordered, for instance, months Jan, Feb, Mar, [...] Dec, lexicographical ordering of strings, or order of data in drop-down control.

3 Criteria of Domain Testing Methods

Complexity of a domain testing method is the smallest possible size of the test suite generated by the method.

The essential part of any testing method is adequacy criterion. *Adequacy criterion* explicitly specifies test case selection, determines whether a test set is adequate, and determines observations that should be done during the testing process [26].

Adequacy criteria of domain testing methods can be characterized by three aspects:

- 1) which kind of values to choose for testing, for instance, only boundary values or only representants of equivalence classes of valid values;
- 2) data coverage principle;

- 3) strategy how the chosen values are combined in test cases according to the data coverage principle.

First aspect considers semantical information of test data according to observations of the testing method, for instance, only boundary values are used or OFF points are added, too, whether the invalid values of parameters are examined or not.

Second aspect is combinatorial strategy based on data coverage. Simplest coverage criterion, each-used, does not take into account how selected values of different parameters are combined, while the more complex coverage criteria, such as pair-wise coverage, are concerned with combinations of values of different parameters.

Each-used (also known as 1-wise) coverage requires that every selected value of each parameter is used at least in one test case of the generated test suite [27].

Pair-wise (2-wise) coverage requires that every possible pair selected values of different parameters are included in the test cases of the test suite.

T-wise coverage requires that every possible combination of values of t parameters is included in the test cases of the test suite. N-wise coverage is a special case of *t-wise* coverage where N is the number of parameters.

As coverage degree t grows, the size of the test suite generated by the testing method also grows. Each-used and N-wise coverage are widely applied in domain testing methods.

Third aspect concerns the strategy how combinatorial coverage of chosen data is implemented. For instance, each-used coverage can be achieved in different ways:

- 1) for every chosen boundary value of each parameter, generate its own test case where all other parameters assume nominal values. In this case the size of the test suite will be the sum of the count of selected values for all parameters together;
- 2) all parameters assume boundary values in each test case – the size of the test suite will be equal to the count of selected values of that parameter for which this count is maximal.

The advantage of the second strategy is the smaller size of the test suite; however, if the test case fails, it is very hard for the tester to say why. The advantage of the first strategy is the possibility to suspect of processing of used boundary value because all other parameters assume nominal values. It is called single fault assumption, which states that faults are very rarely the result of the simultaneous occurrence of two or more faults [5].

Two fault assumption means that in test cases two of input variables assume their boundary values while all remaining variables – their nominal values.

Generally, we can speak about N-fault assumption or multiple fault assumption when the method requires a boundary value for all of program's input variables for the test case.

If the test suite is generated according to single fault assumption and some test case fails, there is a good reason to suppose that the input parameter with boundary value is incorrectly processed. But such test suite will be bigger or more complex than a test suite generated according to N-fault assumption. On the other hand, if the test case generated according to N-fault assumption fails, it is a more complex task to say which boundary or boundaries were processed incorrectly.

4 Complexity of Equivalence Class Testing

Let us examine domain testing methods described in related works. There are two groups of domain testing methods – equivalence class testing methods and boundary value testing methods. If in related works only the methods how to pick values for tests are described, but there are no rules given regarding combination of the selected values in test cases, the author of this paper proposes that a test case is derived for each of the selected values and the other input parameters of the test case assume some valid nominal value.

For the sake of comprehensible drawings, the discussion relates to a program Z with two input parameters X_1 and X_2 . Parameter's X_1 domain is interval $[x_{1a}, x_{1d}]$, which is divided into three equivalence classes $[x_{1a}, x_{1b})$, $[x_{1b}, x_{1c})$, and $[x_{1c}, x_{1d}]$. Parameters' X_2 domain is interval $[x_{2u}, x_{2z}]$, which is divided into two equivalence classes $[x_{2u}, x_{2v})$ and $[x_{2v}, x_{2z}]$. There are invalid values outside domain equivalence classes. For instance, for parameter X_1 such classes are (∞, x_{1a}) and (x_{1d}, ∞) , similarly, for $X_2 - (\infty, x_{2u})$ and (x_{2z}, ∞) .

If the infinite values are boundaries for an equivalence class of valid values, they should be treated as finite – maximal or minimal values that the computer can use for an appropriate data type. The equivalence classes of incorrect values have no boundary values [1].

Weak Equivalence Class Testing

The weak equivalence class testing method examines one representant from each equivalence class of valid values of each parameter [2, 5, 28–32, 42]. The basis of the method is single fault assumption. It is assumed that it is enough to test once each equivalence class.

For our sample program Z , weak equivalence class testing method generates three test cases, for instance as shown in Fig. 3 (a), because the domain of parameter X_1 has the biggest count of equivalence classes – 3.

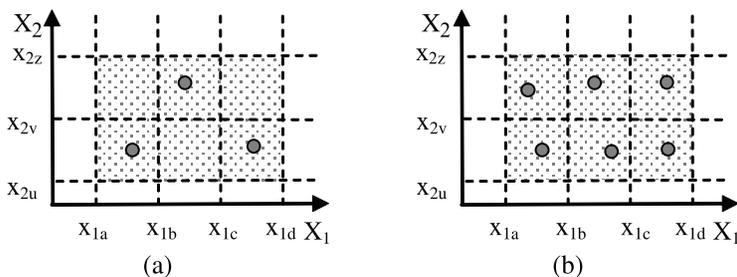


Fig. 3. Test cases for (a) weak equivalence class testing and (b) strong equivalence class testing

In general, for the program with N parameters, the size of the test suite according to the weak equivalence class testing method is $\max_{i=1}^N (M_i)$.

Strong Equivalence Class Testing

The strong equivalence class testing method [5, 33] is based on multiple fault assumption. The test case from each element of the Cartesian product of equivalence classes is included in the test suite, for instance, as shown in Fig. 3 (b).

The strong equivalence class testing method allows to reach two aims – cover all equivalence classes and examine all combinations of different inputs.

In general case, the size of the test suite is $\prod_{i=1}^N M_i$.

Robust Weak Equivalence Class Testing

Robust weak equivalence class testing is similar to weak equivalence testing. In addition, it also considers equivalence classes of invalid values [1, 2, 5, 28–31, 34–38, 42] according to the following algorithm:

- 1) for valid values choose only one value from each equivalence class; furthermore, all parameters have valid values in each test case;
- 2) for invalid values choose one value from each equivalence class and in each test case combine one invalid value with all other valid values. Invalid values for two or more input parameters of the program in the same test case are not allowed (see Fig. 4 (a)).

In N parameters' case, the count of generated test cases is $\max_{i=1}^N (M_i) + \sum_{i=1}^N Q_i$, where Q_i is the size of the set of equivalence classes of invalid values for parameter X_i .

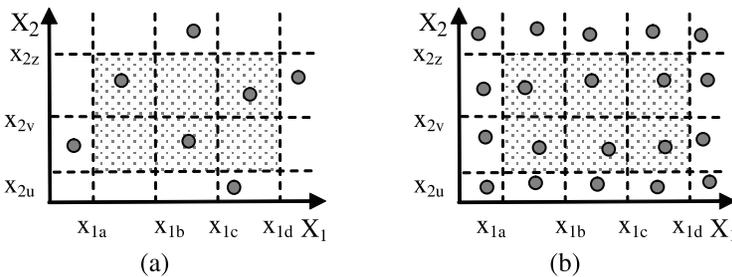


Fig. 4. Test cases for (a) robust weak equivalence class testing and (b) robust strong equivalence class testing

Robust Strong Equivalence Class Testing

The robust strong equivalence class testing [5, 36, 39] method includes in the test suite a test case from each element of Cartesian product of all equivalence classes of valid and invalid values of all parameters, like it is shown in Fig. 4 (b).

In N parameters' case, the count of generated testcases is $\prod_{i=1}^N (M_i + Q_i)$.

Robust Mixed Equivalence Class Testing

Robust mixed equivalence class testing [1] method includes in the test suite a test case from each element of Cartesian product of all equivalence classes of valid values. For invalid values, choose one value from each equivalence class and in each test case

combine one invalid value with all other valid values. Invalid values for two or more input parameters of the program in the same test case are not allowed (see Fig. 5).

In N parameters' case, the count of generated testcases is $\prod_{i=1}^N M_i + \sum_{i=1}^N Q_i$.

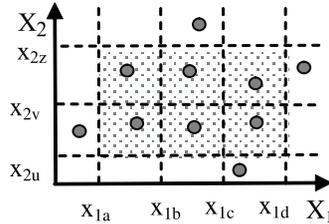


Fig. 5. Test cases for robust mixed equivalence class testing

5 Complexity of Boundary Value Testing Methods

Weak IN Boundary Value Testing

The weak IN boundary value testing method [1, 5, 33, 35] examines boundary values of equivalence classes, inner OFF points, and nominal values. This method is based on single fault assumption – one of the parameters assumes examinable values while all other parameters assume nominal values, like it is shown in Fig. 6 (a) for a 2-dimensional case.

Let us calculate the complexity of the method.

If the program has only one parameter X_1 , 5 test cases are obtained for each equivalence class of valid values – min, min+, nom, max-, max. Because the count of equivalence classes is M_1 , the count of test cases in the test suite is $5M_1$.

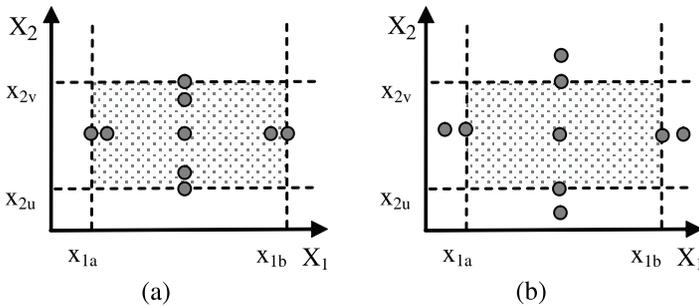


Fig. 6. Test cases for (a) weak IN boundary value testing and (b) weak OUT boundary value testing

If the program has two parameters, test cases are generated according to the following algorithm.

- 1) Hold nominal value of the first equivalence class for the first parameter and obtain 4 test cases by changing the min, min+, max-, max points for the first class of the second parameter.

2) repeat step 1 for each equivalence class of X_2 .

There are $4M_2$ test cases obtained so far.

3) Optimize the test suite – exclude redundant test cases raised by overlapped boundary values of adjacent equivalence classes.

Now we have $4M_2 - L_2$ test cases in our test suite.

4) Repeat steps 1–3 for each equivalence class of parameter X_1 .

There are $(4M_2 - L_2)M_1$ test cases obtained so far.

5) Repeat steps 1–4 with parameters in exchanged roles.

There are $(4M_1 - L_1)M_2 + (4M_2 - L_2)M_1$ testcases obtained during steps 1–5.

6) Now we have to add point test cases when both parameters have nominal values for all elements of Cartesian product of valid equivalence classes of both parameters.

So, we obtain M_1M_2 test cases in this step.

The size of the resulting test suite is:

$$(4M_1 - L_1)M_2 + (4M_2 - L_2)M_1 + M_1M_2 = 9M_1M_2 - L_1M_2 - L_2M_1.$$

There will be $(4N + 1) \prod_{i=1}^N M_i - \sum_{i=1}^N (L_i \prod_{\substack{j=1 \\ j \neq i}}^N M_j)$ test cases in the test suite generated

by the weak IN boundary testing method for the program with N parameters. This is a lower bound of complexity of the weak IN boundary testing method.

If we skip step 3, we obtain that weak IN boundary testing will give no more than test cases. It is the upper bound of complexity of the weak IN boundary testing method, which is achievable in cases when there are no overlapped boundary values between equivalence classes.

Weak OUT Boundary Value Testing

The weak OUT boundary value testing method examines boundary values of equivalence classes, outer OFF points, and nominal case as shown in Fig. 6 (b) [30, 34]. It is very similar to the weak IN boundary value testing method. The only difference is that the weak IN boundary value testing method uses inner OFF points while the weak OUT boundary value testing method uses outer OFF points.

The size of the generated test suite is the same as for the weak IN boundary value testing method – there will be no less than $(4N + 1) \prod_{i=1}^N M_i - \sum_{i=1}^N (L_i \prod_{\substack{j=1 \\ j \neq i}}^N M_j)$ test cases and no more than $(4N + 1) \prod_{i=1}^N M_i$ test cases.

Weak Simple OUT Boundary Value Testing

The weak simple OUT boundary value testing method examines boundary values of equivalence classes and outer OFF points [2, 30, 31, 33, 34, 40]. The only difference from the weak OUT boundary value testing method is that it uses nominal points while the weak simple OUT boundary value testing method does not (Fig. 7 (a)).

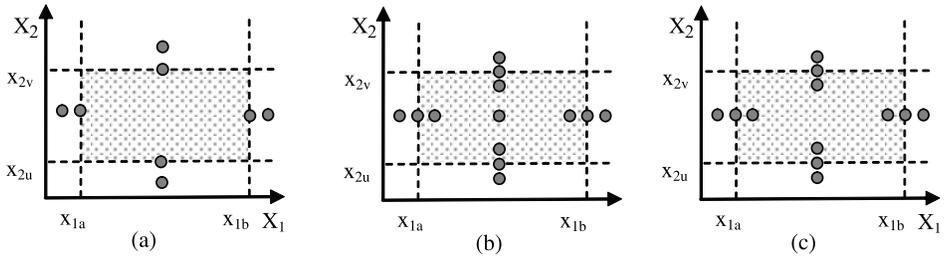


Fig. 7. Test cases for (a) weak simple OUT boundary testing, (b) robust weak boundary value testing and (c) robust weak simple boundary value testing

The size of the generated test suite is smaller than for the weak IN boundary value testing method or the weak OUT boundary value testing method – the lower bound of complexity is $4N \prod_{i=1}^N M_i - \sum_{i=1}^N (L_i \prod_{\substack{j=1 \\ j \neq i}}^N M_j)$ test cases while the upper bound – $4N \prod_{i=1}^N M_i$ test cases.

Robust Weak Boundary Value Testing

The robust weak boundary value testing method [2, 5, 34–36, 38, 42] examines boundary values of equivalence classes, inner and outer OFF points, nominal case (Fig. 7 (b)).

The size of the generated test suite can be obtained similarly to the weak IN boundary value testing method – there will be no less than $(6N + 1) \prod_{i=1}^N M_i - \sum_{i=1}^N (L_i \prod_{\substack{j=1 \\ j \neq i}}^N M_j)$ test cases and no more than $(6N + 1) \prod_{i=1}^N M_i$ test cases.

Robust Weak Simple Boundary Value Testing

The robust weak simple boundary value testing method [2, 42] examines boundary values of equivalence classes, inner and outer OFF points, but unlike the robust weak boundary value testing, it does not test nominal case (Fig. 7 (c)).

The size of the generated test suite can be obtained similarly to the weak IN boundary value testing method – there will be no less than $6N \prod_{i=1}^N M_i - \sum_{i=1}^N (L_i \prod_{\substack{j=1 \\ j \neq i}}^N M_j)$ test cases and no more than $6N \prod_{i=1}^N M_i$ test cases.

Worst Case Boundary Value Testing

The worst case boundary value testing method [5, 41] tests boundary values, inner OFF points, and nominal point. While weak IN boundary value testing used single fault assumption only, the worst case boundary value testing method also uses multiple fault assumption. It means that the method checks what happens if one or all parameters of the program assume special values (Fig. 8 (a)).

If the program has only one parameter X_1 , we obtain 5 test cases for each equivalence class. After that redundant test cases raised by overlapped boundary values of adjacent equivalence classes should be excluded.

So, there are $5M_1 - L_1$ test cases in one parameter's case.

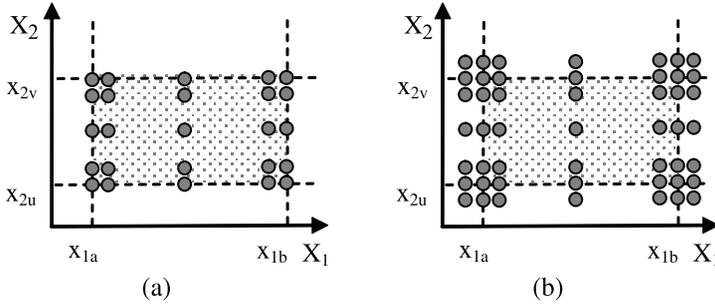


Fig. 8. Test cases for (a) worst case boundary testing and (b) robust worst case boundary testing

If the program has two parameters, test cases are generated according to the following algorithm.

- 1) Hold minimal value of the first equivalence class for the first parameter and obtain 5 test cases by changing the min, min+, max, nom, max+ points for the first class of the second parameter.
- 2) Repeat step 1 for min+, max, nom, max+ of the first equivalence class of X_1 . There are 5×5 test cases obtained so far.
- 3) Repeat the steps 1–2 for each equivalence class of X_1 and optimize the test suite by excluding redundant test cases.

Now we have $5x(5M_1 - L_1)$ test cases in our test suite.

- 4) Repeat steps 1–3 for each equivalence class of parameter X_2 .

There are $(5M_1 - L_1)x(5M_2 - L_2)$ test cases obtained so far.

For N parameters' case, we obtain $\prod_{i=1}^N (5M_i - L_i)$ test cases as the method's lower bound of complexity, but as the upper bound of complexity, the worst case boundary value testing method will give no more than $\prod_{i=1}^N 5M_i = 5^N \prod_{i=1}^N M_i$ test cases.

Robust Worst Case Boundary Value Testing

The robust worst case boundary value testing method [5, 39] tests boundary values, inner and outer OFF points, and nominal point (Fig. 8 (b)).

Acting according to the algorithm that is analogical to the worst case boundary value testing method, we obtain that program with N parameters will have $\prod_{i=1}^N (7M_i - 2L_i)$ test cases or no more than $7^N \prod_{i=1}^N M_i$ test cases.

Weak Corner IN Boundary Value Testing

Weak corner IN boundary value testing complies with multiple fault assumption. It tests cases when all parameters assume the same type of special values – boundary values, inner OFF points, or nominal point as it is shown in Fig. 9 (a) [1, 41].

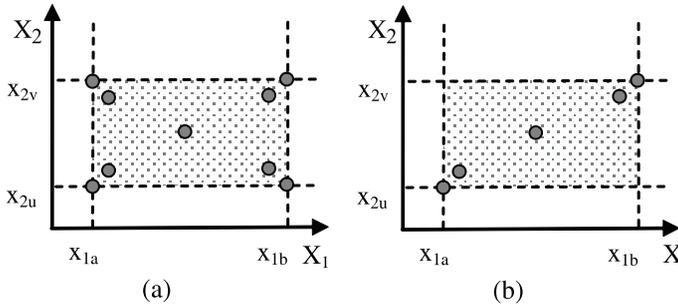


Fig. 9. Test cases for (a) weak corner IN boundary value testing and (b) weak diagonal IN boundary value testing

If the program has two parameters, test cases are generated according to the following algorithm.

- 1) Let us take boundary values of the first equivalence class of both parameters. We obtain 4 test cases (min, min), (min, max), (max, min), and (max, max).
- 2) Let us consider inner OFF points of the first equivalence class of both parameters. We again obtain 4 test cases (min+, min+), (min+, max-), (max-, min+), and (max-, max-).
- 3) Nominal points of the first equivalence class of both parameters will give one test case (nom, nom).

Now we have $2 \times 2^2 + 1$ testcases for the first element in the set of Cartesian product of equivalence classes. For all elements, there will be $(2 \times 2^2 + 1) M_1 M_2$ test cases.

- 4) Now exclude redundant test cases raised by overlapping boundary values and obtain $(2^{2+1} + 1) \prod_{i=1}^2 M_i - 2 \sum_{i=1}^2 (L_i \prod_{\substack{j=1 \\ j \neq i}}^2 M_j)$ test cases.

There will be $(2^{N+1} + 1) \prod_{i=1}^N M_i - 2 \sum_{i=1}^N (L_i \prod_{\substack{j=1 \\ j \neq i}}^N M_j)$ test cases in the test suite generated

by the weak corner IN boundary testing method for the program with N parameters.

If we skip step 4, we obtain that the weak corner IN boundary value testing method will give no more than $(2^{N+1} + 1) \prod_{i=1}^N M_i$ test cases.

Weak Diagonal IN Boundary Value Testing

Weak diagonal IN boundary value testing complies with multiple fault assumption, too. But it tests cases when all parameters assume the same type and meaning of special values – if it is boundary values, all parameters assume minimal values or all assume maximal values, the same goes for inner OFF points or nominal point as it is shown in Fig. 9 (b) [41].

If the program has two parameters, test cases are generated according to the following algorithm.

- 1) Let us take boundary values of the first equivalence class of both parameters. We obtain 2 test cases (min, min), (max, max).
- 2) Let us consider inner OFF points of the first equivalence class of both parameters. We again obtain 2 test cases (min+, min+) and (max-, max-).
- 3) Nominal points of the first equivalence class of both parameters will give one test case (nom, nom).

Now we have 5 test cases for the first element in the set of Cartesian product of equivalence classes. For all elements, there will be $5M_1M_2$ test cases.

- 4) Now exclude redundant test cases raised by overlapping boundary values and obtain $5M_1M_2 - L_1L_2$ test cases.

There will be no less than $5\prod_{i=1}^N M_i - \prod_{i=1}^N L_i$ test cases in the test suite generated by the weak diagonal IN boundary testing method for the program with N parameters.

If we skip step 4, we obtain that the weak diagonal IN boundary value testing method will give no more than $5\prod_{i=1}^N M_i$ test cases.

Multidimensional Boundary Value Testing

The multidimensional boundary value testing method requires at least one test case for each boundary value of each equivalence class (Fig. 10 (a)) [1, 2, 5, 28–30, 32, 33].

In the case of N parameters, there will be $\max_{i=1}^N (2M_i - L_i)$ test cases or no more than $\max_{i=1}^N (2M_i)$ test cases if we do not have overlapping boundaries.

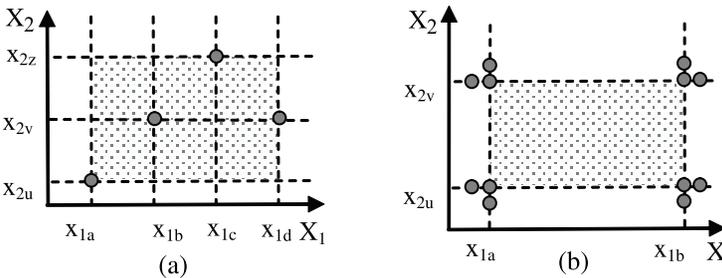


Fig. 10. Test cases for (a) multidimensional boundary value testing and (b) robust corner OUT boundary value testing

Robust Corner OUT Boundary Value Testing

The robust corner OUT boundary value testing method tests cases when parameters assume boundary values or outer OFF points as it is shown in Fig. 10 (b) [42].

Let us count the test cases where one of inputs is OFF point.

If the program has only one parameter X_1 , 2 test cases are obtained for each equivalence class of valid values – min- and max+. Because the count of equivalence classes is M_1 , the count of test cases in the test suite is $2M_1$.

If the program has two parameters, test cases are generated according to the following algorithm.

- 1) We obtain $2M_1$ test cases on each boundary of the second parameter, obtaining $2M_1(2M_2 - L_2)$ test cases.
- 2) Repeat step 1 for each boundary class of X_1 and obtain $2M_2(2M_1 - L_1)$.

Generalize this to N parameters – $\sum_{i=1}^N (2M_i \prod_{\substack{j=1 \\ j \neq i}}^N (2M_j - L_j))$.

The count of the test cases where all inputs are boundary values is $\prod_{i=1}^N 2M_i$.

So there will be at least $\prod_{i=1}^N 2M_i + \sum_{i=1}^N (2M_i \prod_{\substack{j=1 \\ j \neq i}}^N (2M_j - L_j))$ test cases in the test suite

generated by the robust corner OUT boundary testing method for the program with N parameters.

The upper bound of the method's complexity is

$$\prod_{i=1}^N 2M_i + \sum_{i=1}^N (2M_i \prod_{\substack{j=1 \\ j \neq i}}^N 2M_j) = 2^N \prod_{i=1}^N M_i + \sum_{i=1}^N (2^N \prod_{j=1}^N M_j) = 2^N (N + 1) \prod_{i=1}^N M_i .$$

Robust Strong Boundary Value Testing

The robust strong boundary value testing method [1] takes all boundary values, inner OFF values, and outer OFF values of equivalence classes of valid values and divides them into two sets – all valid values in one set and invalid values in the other set (Fig. 11).

The method allows to combine freely all values from the set of valid values in test cases.

The set of invalid values is revised. If there are two or more values from the same equivalence class, only one of them is left in the set. The method requires exactly one test case for each value that is left in the set where all the other values in the test case are valid values.

For each parameter, we have L_i overlapping borders. They will give $3L_i$ valid values. We also have $2M_i - 2L_i$ non-overlapping borders which will give exactly $2M_i - 2L_i$ invalid values – outer OFF points that fall into classes of invalid values, and exactly $2M_i - 2L_i$ valid values – inner OFF points of equivalence classes of valid values.

Hence, according to the method's adequacy criterion, valid values for N parameters will give $\max_{i=1}^N (3L_i + 2M_i - 2L_i) = \max_{i=1}^N (2M_i + L_i)$ test cases, but invalid values $\sum_{i=1}^N (2M_i - 2L_i)$ test cases.

The only question is about exactly $2M_i - 2L_i$ boundary values. They can be valid if each interval of valid equivalence classes is closed or otherwise invalid.

Thus, the lower bound of the method can be achieved when all questionable values

$$\text{are valid: } \max_{i=1}^N (2M_i + L_i) + \sum_{i=1}^N (2M_i - 2L_i) + \max_{i=1}^N (2M_i - 2L_i).$$

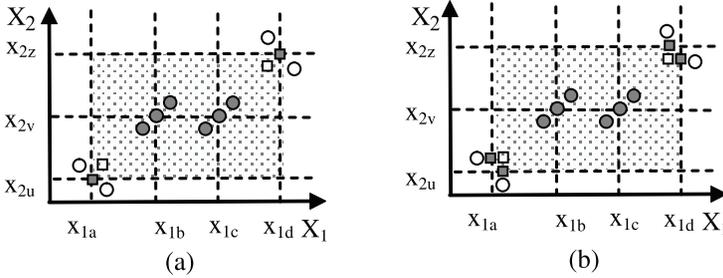


Fig. 11. Test cases for robust strong boundary value testing; (a) shows lower bound case, figure (b) – upper bound case. Filled dots represent test cases of overlapping borders, unfilled squares – test cases of inner OFF points of non-overlapping borders. Unfilled dots represent test cases of outer OFF points of non-overlapping borders. Filled squares show test cases of boundary values of non-overlapping borders when (a) they all are valid values, (b) they all are invalid values.

The upper bound of method can be achieved when all questionable values are invalid:

$$\max_{i=1}^N (2M_i + L_i) + \sum_{i=1}^N (2M_i - 2L_i) + \sum_{i=1}^N (2M_i - 2L_i) = \max_{i=1}^N (2M_i + L_i) + 4 \sum_{i=1}^N (M_i - L_i).$$

6 The Complexity and Subsumption Hierarchy of Domain Testing Methods

The majority of comparisons of testing adequacy criteria in related works use the subsume ordering. It was used to compare data flow testing adequacy criteria and several other structural coverage criteria [43–48]. Other ways of comparing the testing criteria have been studied and compared in [25, 49].

One of the formulations of subsume definition is the following: “let C1 and C2 be two software data adequacy criteria. C1 is said to subsume C2 if for all programs p under test, all specifications s and all test sets t, t is adequate according to C1 for testing p with respect to s implies that t is adequate according to C2 for testing p with respect to s” [49]. In other words, C1 subsumes C2 if every test suite generated by C1 is adequate for C2, too.

The hierarchy of domain testing methods according to subsume relation is showed in Fig. 12. Principles of test case generation for the corresponding criteria of each box are schematically showed in the figure.

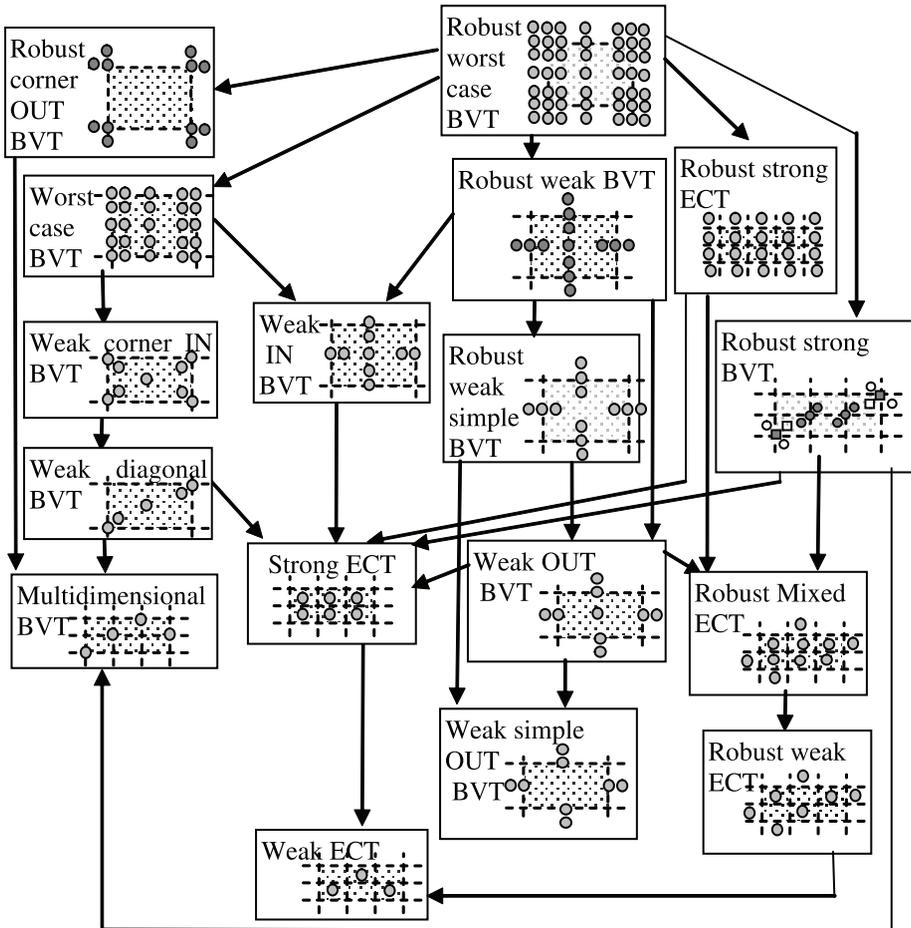


Fig. 12. The hierarchy of domain testing methods according to subsume relation

In most cases the subsumption is immediate and because of the limited space of the paper will not be discussed. Let us look at some cases.

1) Robust worst case BVT > Robust strong ECT

Robust strong ECT criterion requires a test case from each Cartesian product of each parameter's equivalence classes of valid values and equivalence classes of invalid values. Robust worst case BVT provides inner OFF points and nominal point from each Cartesian product of equivalence classes of valid values (points on boundaries are not suitable because there is a possibility that a boundary does not belong to equivalence class) and outer OFF points from Cartesian product in which equivalence classes of invalid values are involved. In the 2-dimensional case, it looks as showed in Fig. 13. Crosses and dots together are the test suite of robust worst case BVT criteria, but crosses alone are an adequate test suite of robust strong ECT criteria. In such a way, from each test suite of robust worst case BVT a test suite of robust strong ECT can be obtained.

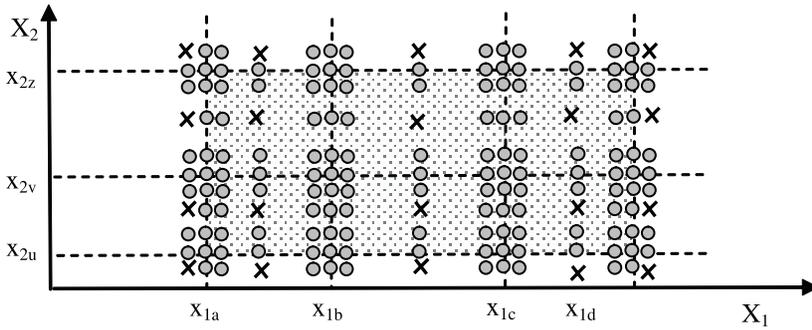


Fig. 13. Crosses and dots together are the test suite of robust worse case BVT criteria, but crosses alone are an adequate test suite of robust strong ECT criteria

2) Weak OUT BVT > Robust mixed ECT

Robust mixed ECT criterion asks for a test case from each element of Cartesian product of equivalence classes of valid values and from each class of invalid values. Weak OUT BVT provides nominal point for each element of Cartesian product of equivalence classes of valid values and outer OFF points for adjacent equivalence classes of invalid values. In the 2-dimensional case, it looks as showed in Fig. 14. Crosses and dots together are the test suite of weak OUT BVT criteria, but crosses alone are one of the adequate test suites of robust mixed ECT criteria. In such a way, from each test suite of weak OUT BVT a test suite of robust mixed ECT can be obtained.

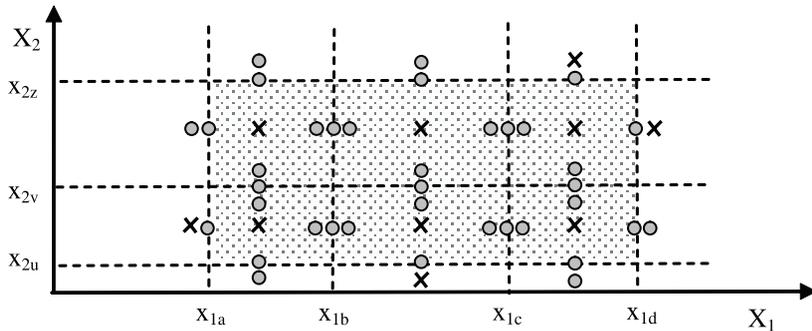


Fig. 14. Crosses and dots together are the test suite of weak OUT BVT criteria, but crosses alone are an adequate test suite of robust mixed ECT criteria

3) Weak simple OUT BVT does not subsume robust weak ECT.

Consider the situation in Fig. 15. Parameter X_1 has one equivalence class for valid values with boundaries that do not belong to it (x_{1a} , x_{1b}) and parameter X_2 has the same situation – equivalence class (x_{2u} , x_{2v}). Weak simple OUT BVT cannot provide a test case when both parameters assume valid values which is required by robust weak ECT.

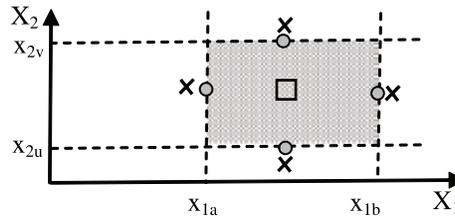


Fig. 15. Crosses and dots together are the testsuite of weak simple OUT BVT criteria. Crosses alone are candidates for the test suite of robust weak ECT criteria, while square represents the test case required by robust weak ECT that weak simple OUT BVT criteria cannot provide.

Table summarizes the lower and upper bounds of complexity of domain testing methods. It is easy to see that the robust worst case BVT method which is at the top of subsumption hierarchy in Fig. 12 also has the highest assessments of complexity. The methods in the lowest level of hierarchy have the lowest assessments of complexity.

It is obvious that if an adequacy criterion of method C1 subsumes an adequacy criterion of method C2, complexity of C1 is higher than of C2 of domain testing methods and their data adequacy criteria.

However, we cannot conclude that if method’s C1 complexity is higher than method’s C2 complexity, C1 subsumes C2. For instance, the complexity of the weak simple OUT BVT method is higher than the complexity of the weak ECT method. But the weak simple OUT BVT method tests a completely other kind of points than the weak ECT method; hence, the weak simple OUT BVT method does not subsume the weak ECT method.

Table

The summarizing table of complexity of domain testing methods, where N – count of program’s parameters, M_i – size of the set of equivalence classes of valid values for parameter X_i , Q_i – size of the set of equivalence classes of invalid values for parameter X_i , and L_i – size of the set of all common boundary values for parameter X_i

No.	Method	Complexity	
		Lower bound	Upper bound
1	Weak equivalence class testing	N $\max_{i=1}^N (M_i)$	The same as lower bound
2	Strong equivalence class testing	$\prod_{i=1}^N M_i$	The same as lower bound
3	Robust weak equivalence class testing	$\max_{i=1}^N (M_i) + \sum_{i=1}^N Q_i$	The same as lower bound
4	Robust strong equivalence class testing	$\prod_{i=1}^N (M_i + Q_i)$	The same as lower bound
5	Robust mixed equivalence class Testing	$\prod_{i=1}^N M_i + \sum_{i=1}^N Q_i$	The same as lower bound

6	Weak IN boundary value testing	$(4N+1)\prod_{i=1}^N M_i - \sum_{i=1}^N (L_i \prod_{\substack{j=1 \\ j \neq i}}^N M_j)$	$(4N+1)\prod_{i=1}^N M_i$
7	Weak OUT boundary value testing	$(4N+1)\prod_{i=1}^N M_i - \sum_{i=1}^N (L_i \prod_{\substack{j=1 \\ j \neq i}}^N M_j)$	$(4N+1)\prod_{i=1}^N M_i$
8	Weak simple OUT boundary value testing	$4N\prod_{i=1}^N M_i - \sum_{i=1}^N (L_i \prod_{\substack{j=1 \\ j \neq i}}^N M_j)$	$4N\prod_{i=1}^N M_i$
9	Robust weak boundary value testing	$(6N+1)\prod_{i=1}^N M_i - 2\sum_{i=1}^N (L_i \prod_{\substack{j=1 \\ j \neq i}}^N M_j)$	$(6N+1)\prod_{i=1}^N M_i$
10	Robust weak simple boundary value testing	$6N\prod_{i=1}^N M_i - 2\sum_{i=1}^N (L_i \prod_{\substack{j=1 \\ j \neq i}}^N M_j)$	$6N\prod_{i=1}^N M_i$
11	Worst case boundary value testing	$\prod_{i=1}^N (5M_i - L_i)$	$5^N \prod_{i=1}^N M_i$
12	Robust worst case boundary value testing	$\prod_{i=1}^N (7M_i - 2L_i)$	$7^N \prod_{i=1}^N M_i$
13	Weak corner IN boundary value testing	$(2^{N+1} + 1)\prod_{i=1}^N M_i - 2\sum_{i=1}^N (L_i \prod_{\substack{j=1 \\ j \neq i}}^N M_j)$	$(2^{N+1} + 1)\prod_{i=1}^N M_i$
14	Weak diagonal IN boundary value testing	$5\prod_{i=1}^N M_i - \prod_{i=1}^N L_i$	$5\prod_{i=1}^N M_i$
15	Multidimensional boundary value testing	$\max_{i=1}^N (2M_i - L_i)$	$2 \max_{i=1}^N (M_i)$
16	Robust corner OUT boundary value testing	$\prod_{i=1}^N 2M_i + \sum_{i=1}^N (2M_i \prod_{\substack{j=1 \\ j \neq i}}^N (2M_j - L_j))$	$2^N (N+1)\prod_{i=1}^N M_i$
17	Robust strong boundary value testing	$\max_{i=1}^N (2M_i + L_i) + \sum_{i=1}^N (2M_i - 2L_i) + \max_{i=1}^N (2M_i - 2L_i)$	$2 \max_{i=1}^N (M_i + L_i) + 2\sum_{i=1}^N (2M_i - L_i)$

7 Conclusions and Future Directions

Equivalence partitioning and boundary value analysis techniques are frequently mentioned in the books about software testing. In most cases, they describe common principles how to derive equivalence classes, boundary values, and present some

strategies how to make test cases from them. Each author or authors often emphasize one or two of the strategies but do not analyze their weaknesses or strengths, just demonstrate some examples. The positive exception is Jorgensen in [5].

This survey is an attempt to collect and describe some equivalence class testing and boundary value testing methods frequently mentioned in literature in a comprehensive manner. This paper assesses and summarizes complexity of domain testing methods and provides the hierarchy of domain testing methods according to subsume ordering. The complexity of testing methods is treated from the aspect of the size of the test suite generated by the method.

The author concludes, if some domain testing method subsumes other domain testing method, the first method also has higher complexity than the second method. However, the reverse statement is invalid.

This paper analyses the adequacy criteria of domain testing methods from three aspects – the kind of values to choose for testing, the data coverage principle, and the strategy how the chosen values are combined in test cases according to the data coverage principle.

Analysis of literature shows that methods with the smallest complexity are described in most cases. It shows that in praxis the most applicable are the methods that produce a not-too-big size of the test suite and are also practical – the methods that use single fault assumption in order to diagnose easier the cause of failure if it occurs.

Despite many sources of literature that cover boundary value analysis and equivalence partitioning, several important issues still remain largely unexplored. One of such issues is how the effectiveness and efficiency of testing is affected by the choice of data coverage criteria and the coverage of combinations of special or boundary values of different parameters of software.

The second issue related to the use of domain testing methods that is not adequately investigated is how to test the mutually dependent parameters of software better.

The third issue that needs further investigation is how the various models that describe the behavior of software can be used with the aim to refine equivalence classes of software input domain.

References

1. Spillner A., Linz T., Schaefer H. (2007) *Software Testing Foundations. A Study Guide for the Certified Tester Exam*. Foundation Level, ISTQB compliant. Rocky Nook Inc.
2. Veenendaal E. (2002) *The Testing Practitioner*. UTN Publishers, Den Bosch, NL.
3. Beizer B. (1990) *Software Testing Techniques*. 2nd ed. New York: Van Nostrand Reinhold.
4. Beizer B. (1995) *Black Box Testing*. New York: John Wiley.
5. Jorgensen P. C. (1995) *Software Testing: A Craftman's Approach*. Boca Raton, London, New York, Washington D.C.: CRC Press.
6. Howden W. E. (1976) Reliability of the Path Analysis Testing Strategy. *IEEE Trans. Softw. Eng.* vol. 2, no. 3, 208–215.
7. Barzdins J., Bicevskis J., Kalnins A. (1975) Construction of complete sample system for correctness testing. In: Proc. MFCS 1975. LNCS, vol. 32, Berlin / Heidelberg: Springer, pp. 1–12.
8. Howden, W. E. (1975) Methodology for the Generation of Program Test Data. *IEEE Transactions on Computers*, vol. 24, no. 5, 554–560.
9. Clarke L. A. (1976) A System to Generate Test Data and Symbolically Execute Programs. *IEEE Trans. on Softw. Eng.* vol. 2, no. 3, 215–222.

10. Bičevskis J., Borzovs J., Straujums U., Zariņš A., Miller E. F. (1979) SMOTL – A System to Construct Samples for Data Processing Program Debugging. *IEEE Trans. Softw. Eng.* 5, 1, 60–66.
11. Richardson D. J., Clarke L. A. (1981) A partition analysis method to increase program reliability. In: Proceedings of the 5th international Conference on Software Engineering, IEEE Press, Piscataway, NJ, 244–253.
12. Richardson D. J., Clarke L. A. (1985) Partition analysis: a method combining testing and verification. *IEEE Trans. Softw. Eng.* SE-11, 12, 1477–1490.
13. Jeng B., Weyuker E. (1989) Some observations on partition testing. In: Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification, ACM, New York, NY, 38–47.
14. Hamlet D., Taylor R. (1990) Partition Testing Does Not Inspire Confidence (Program Testing). *IEEE Trans. Softw. Eng.* 16, 12, 1402–1411.
15. Weyuker E. J., Jeng B. (1991) Analyzing Partition Testing Strategies. *IEEE Trans. Softw. Eng.* 17, 7, 703–711.
17. De Millo R., McCracken W. M., Martin R. J., Passafiume J. (1987) *Software Testing and Evaluation*. Benjamin-Cummings Publishing Co., Inc.
18. White L. J., Cohen E. I. (1980) A Domain Strategy for Computer Program Testing. *IEEE Trans. Softw. Eng.* 6, 3, 247–257.
19. Clarke L. A., Hassell J., Richardson D. J. (1982) A Close Look at Domain Testing. *IEEE Trans. Softw. Eng.* 8, 4, 380–390.
20. Afifi F. H., White L. J., and Zeil S. J. (1992) Testing for linear errors in nonlinear computer programs. In: Proceedings of the 14th international Conference on Software Engineering ICSE '92. ACM, New York, 81–91.
21. Jeng B., Weyuker E. J. (1994) A simplified domain-testing strategy. *ACM Trans. Softw. Eng. Methodol.* 3, 3, 254–270.
22. Zeil S. J., Afifi F. H., (1992) White L. J. Detection of linear errors via domain testing. *ACM Trans. Softw. Eng. Methodol.* 1, 4, 422–451.
23. Hajnal Á., Forgács I. (1998) An applicable test data generation algorithm for domain errors. In: Tracz W. (ed.) Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '98. ACM, New York, NY, 63–72.
24. Chien Y., Liu S. (2004) An Approach to Detecting Domain Errors Using Formal Specification-Based Testing. In: Proceedings of the 11th Asia-Pacific Software Engineering Conference, APSEC. IEEE Computer Society, Washington, DC, 276–283.
25. Weyuker E. J., Weiss S. N., Hamlet D. (1991) Comparison of program testing strategies. In: Proceedings of the Symposium on Testing, Analysis, and Verification, TAV4. ACM, New York, NY, 1–10.
26. Zhu H., Hall P. A., May J. H. (1997) Software unit test coverage and adequacy. *ACM Comput. Surv.* 29, 4, 366–427.
27. Grindal M., Offutt J., Andler S. F. (2005) Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, 15, 3, 167–199.
28. Kaner C., Bach J., Pettichord B. (2002) *Lessons Learned in Software Testing: A Context Driven Approach*. New York, Chichester, Weinheim, Brisbane, Singapore, Toronto: John Wiley & Sons, Inc.
29. Culbertson R., Brown C., Cobb G. (2001) *Rapid Testing*. Prentice Hall PTR.
30. Kaner C., Falk J., Nguyen H. Q. (1999) *Testing Computer Software*. 2nd ed. New York, Chichester, Weinheim, Brisbane, Singapore, Toronto: John Wiley & Sons, Inc.
31. Myers G. J. (2004) *The Art of Software Testing*. 2nd ed., Hoboken, New Jersey: John Wiley & Sons, Inc.
32. Nguyen H. Q., Johnson B., Hackett M. (2001) *Testing Applications on the Web: Test Planning for Mobile and Internet-Based Systems*. 2nd ed., John Wiley & Sons, Inc.
33. McGregor J. D., Sykes D. A. (2001) *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc.
34. Broekman B., Notenboom E. (2003) *Testing Embedded Software*. Addison-Wesley.
35. Link J., Frolich P. (2003) *Unit Testing in Java: how Tests Drive the Code*. Morgan Kaufmann Publishers Inc.
36. Watkins J. (2001) *Testing IT: an Off-the-Shelf Software Testing Process*. Cambridge, United Kingdom: Cambridge University Press.
37. Perry W. E. (2006) *Effective Methods for Software Testing*. 3rd ed. Hungry Minds Inc.
38. Craig R. D., Jaskiel S. P. (2002) *Systematic Software Testing*. Boston / London: Artech House Publishers.

39. Dustin E. (2002) *Effective Software Testing: 50 Ways to Improve Your Software Testing*. Addison-Wesley Longman Publishing Co., Inc.
40. Everett G. D., McLeod R., Jr. (2007) *Software Testing: Testing Across the Entire Software Development Life Cycle*. Wiley-IEEE Computer Society Press.
41. Gao J. Z., Tsao J., Wu Y., Jacob T. H. (2003) *Testing and Quality Assurance for Component-Based Software*. Artech House, Inc.
42. Copeland L. (2003) *A Practitioner's Guide to Software Test Design*. Artech House, Inc.
43. Rapps S., Weyuker E. J. (1985) Selecting software test data using data flow information. *IEEE Trans. Softw. Eng. SE*, 11, 4, 367–375.
44. Clarke L. A., Podgurski A., Richardson D., Zeil S. (1985) A comparison of data flow path selection criteria. In: Proc. 8th ICSE, 244–251.
45. Frankl P. G., Weyuker J. E. (1988) An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng. SE*, 14, 10, 1483–1498.
46. Ntafos S. C. (1988) A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng. SE*, 14, 868–874.
47. Weiser M. D., Gannon J. D., McMullin P. R. (1985) Comparison of structured test coverage metrics. *IEEE Software*, 2, 2, 80–85.
48. Weiss S. N. (1989) Comparing test data adequacy criteria. *SIGSOFT Softw. Eng. Notes* 14, 6, 42–49.
49. Hamlet R. (1989) Theoretical comparison of testing methods. In: Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification, TAV3. ACM, New York, 28–37.