A systematic Literature Review to Classify Pre and Post Test Suite Reduction Techniques

Mohammed Akour, Iyad Alazzam, Feras Hanandeh and Iman Akour

Abstract— Test suite reduction is a critical activity which takes a place before or after test cases generation process. As software keeps growing large amounts of new test cases will be generated and added to the test pool and others will be updated, accordingly test suite size will keep increasing. Test suite reduction techniques have been proposed to eliminate redundant or irrelevant test cases based on variant criteria, while seeking to maintain the total effectiveness of the reduced test suite. This paper presents a systematic literature review to classify some existing techniques and perform sort of comparison in terms of pros and cons. A major result of this paper is a categorization of the test suite reduction which could provide a guideline for software testers in choosing the best technique based on the test requirements.

Keywords— Systematic Literature Review; Test suite reduction techniques.

I. INTRODUCTION

One of the most important phases of SDLC (Software development life cycle) is Software Testing. It is an important component of software quality assurance. There are many definitions available for Software Testing, but one can shortly define that as: A process of executing a program with goal of finding errors [2]. Some people get confused about the goal of testing, thinking that the goal is to check if a program is free from errors, while the goal is finding errors. So tests show the presence not the absence of defects. Miller gives a good description of testing in [3]: “The general aim of testing is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances”.

Dr. IyadAlazzam, Computer Information Systems, Yarmouk University, Jordan, eyadh@yu.edu.jo

Dr. Mohammed Akour, Computer Information Systems, Yarmouk University, Jordan, mohammed.akour@yu.edu.jo

Dr. Feras Hanandeh, Computer Information System, Hashemite University, Jordan, feras@hu.edu.jo

Dr. Iman Akour, University of Sharja, iakour@sharjah.ac.ae

Testing typically consumes 40–50% of development efforts, and consumes more effort for systems that require higher levels of reliability [4]. Although it is often impossible to find all errors in the program, the selection of right strategy at the right time will make the software testing efficient and effective [5].

The tester may or may not know the inside details of the software module under test, therefore either white-box testing or black-box testing can be used against the software module by generating a set of test cases [6]. A set of test cases is a set of (inputs, execution preconditions, and expected outcomes).

This means that test cases check if a program for specified inputs gives the expected results. While a Test-Suite is a set of requirements and subsets of test cases, each requirement must be satisfied be at least one test case [1].

Our paper is organized as follows: in section 2, we present the problem under investigation. Section 3 demonstrates the related works. Section 4 gives details about the systematic review process and its application. In section 5, we describe each reduction technique and provide a comparison between them, and finally we conclude the paper in section 6.

II. PROBLEM UNDER INVESTIGATION

With a tremendous number of possible test cases available, especially in case of complex programs, testers have to generate appropriate test cases in a way that reduces the cost, time and efforts of executing and validating tests [8]. Another important aspect of software testing is the number of test cases that have a direct effect on the cost of testing, particularly that of regression testing [7] (testing activity that is performed to provide confidence that the changes made don’t harm the existing behavior of the software), it means the process of retesting the software after changes. So when tests must be run repeatedly for every change in the program, it is advantageous to have as small set of test cases as possible. Thus test case reduction aims to finding a minimal subset of the test-suite that can cover all requirements [7]. Many techniques are available and have their own advantages and disadvantages and we can classify them into two types:

- Pre-process Reduction techniques (techniques reduce the test-suite before generation).
- Post-process Reduction techniques (techniques reduce the test-suite after generation).

The main goal of this article is to expose some of available pre and post test case reduction techniques and briefly manifest the mechanism for each technique. We compare these techniques by considering their advantages and disadvantages.

III. RELATED WORKS

Test suite minimization techniques (post-process) reduce the size of the test suite based on removing redundant test cases(unnecessary test cases) from it. There are many researchers who proposed a method to reduce unnecessary test cases, like Rothermel [17], McMaster [18] and Sampath [19]. These techniques intend to get rid of and minimize a size of test cases while maintaining the ability to detect faults. Previous works on test case minimization can be regarded as the development of different heuristics for the minimal hitting set problem. Horgan and London applied linear programming to the test case minimization problem in their implementation of a data-flow based testing tool, ATAC [21, 22]. Akour et al [33] provide test case reduction technique for adaptive software system. Their approach employed Change propagation theme to synchronizing component models and runtime test models and then removed the test cases that associated with a component targeted in reductive changes.

Employing model-checker facilitates the detection of equivalent mutants. Therefore, only non-equivalent mutants are used for the evaluation of a mutant score. Heimdahl and Devaraj [25] proposed a minimization approach which is applied to the model-checker scenario. A reduced subset of the test-suite fulfilling a criterion can be identified by calculating the covered properties for each test-case, and then repetitively picking the test case that covers the most yet uncovered properties. Black [24] proposed a test-case generation approach based on mutation of the reflected transition relation. The mutated, reflected properties can be utilized to catch properties for test-case generation, to specify mutant score and for minimization as well.

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, so the main goal of domain testing methods is to achieve a test suite in which the size is considerably smaller than the count of all inputs of the program, and which effectively reveals failures of the program as much as possible [26]. There are two groups of domain testing methods – equivalence class testing (ECT) methods and boundary value testing (BVT) methods [26].

There are many methods that different authors call domain testing methods or domain analysis methods that take into account dependencies or interactions between input parameters [7, 27, and 28]. 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 [7, 27], but there are some methods that allow to test nonlinear boundaries, too [29, 30].

IV. RESEARCH METHOD

This review included the following steps:

1. Formulate a review protocol.
2. Conduct the review (identify and evaluate primary studies, extract and synthesize data to produce a concrete result).
3. Analyze the results.
4. Report the results.
5. Discuss the findings.

The review protocol specified the questions to be addressed, the databases to be searched and the methods to be used to identify, assemble, and assess the evidence. To reduce researcher bias, the protocol, described in the remainder of this section, was developed by one author, reviewed by another author and then finalized through discussion, review, and iteration among the authors and their research group.

V. RESEARCH QUESTION

The main goal of this systematic review is to identify, estimate and classify the Approaches, techniques, methods, and tools in test suite reduction techniques, to concentrate well on the systematic review, as of research questions are needed. The high-level question addressed by this review is:

What types of techniques and approaches in test suite reduction can be identified from the literature. The high-level research question was decomposed into four specific research questions, which guided the literature review. The first question tries to assess and measure the usefulness and importance of estimation of test case coverage .The second question looks for identify types of test cases that can remove or retain in the test suite reduction and which kinds of test cases are removed more frequently in the test suite reduction process. The third question focus on identifying test suite reduction methods. The final question concerns with the taxonomy of techniques in test suite reduction that will help in selecting which test cases should be removed or retained in the test suite based on its classification and type.

Table 1 – Source List

Databases	Other Journals and Conference	Other Sources
IEEExplore INSPEC ACM Library SCIRUS Science Citation Index	• *Transactions on Autonomous and Adaptive Systems (TAAS)* • *International Conference on Autonomic Computing (ICAC)*	Reference lists from primary studies

Table 2 – Inclusion and Exclusion Criteria

Inclusion Criteria	Exclusion Criteria
Papers that talk dynamic adaptive systems.	Papers that are based only on expert opinion
Papers about testing, validating, verifying for dynamic adaptive systems.	Short papers**,** tutorials, and mini-tracks
Papers about classifications, components risks in adaptive systems.	Studies not related to any of the research questions
Empirical studies (qualitative or quantitative)	Preliminary conference versions of included journal papers
Other papers that directly address the research questions	Studies presented in language other than English
	Studies whose findings are unclear and ambiguous

VI. SOURCE SELECTION AND SEARCH

Prior to conducting the search, the correct set of databases must be selected to optimize the likelihood of finding the most complete and relevant sources. In this review, the following criteria were used to select the source databases:

The databases were chosen to include journals and conference proceedings that cover: test suite reduction, test case selection, test case prioritization, test case prioritization, and empirical studies.

The databases had to have a search engine with an advanced search mechanism that allowed keyword searches;

The list of databases was reduced where possible to minimize the redundancy of journals and proceedings across databases. The final source list appears in Table 1.

Based on the criteria for selecting database sources (mentioned earlier in this section), an initial list of sources was developed. To search these databases, a set of search strings was created for each research question based on keywords extracted from the research questions and augmented with synonyms. In developing the keyword strings to use when searching the source databases, the following principles were applied:

The major terms were extracted from the review questions and augmented with other terms known to be relevant to the research;

A list of meaningful synonyms, abbreviations, and alternate spellings were then generated.

The following global search string was constructed containing all of the relevant keywords and their synonyms:

((suite OR set OR group OR collection) AND (testing OR investigation OR check OR analysis OR inspection OR assessment OR evaluation OR examination OR review OR measurement OR verify OR validate OR authenticate OR confirm OR ensure OR prove) AND (approach OR process OR system OR technique OR methodology OR procedure OR mechanism OR plan OR pattern) AND (type OR taxonomy OR classification OR categorization OR grouping OR organization OR terminology OR systematization) AND (priority OR preference OR primacy OR superiority) AND (test OR check OR examination OR assessment AND (test case) AND (policy OR strategy OR plan OR guidelines OR rule) AND (reduction OR decrease OR decline OR cut OR drop OR lessening)

Using this global search string, five different search strings (each one with its own purpose) were derived and executed on each database. Executing the search strings on the databases in Table 2 resulted in an extensive list of potential papers that could be included in the review. To ensure that only the most relevant papers were included a set of detailed inclusion and exclusion criteria are shown in table 2.

Using these criteria, the results of the database searches were examined to arrive at the final list of papers. The process followed for paring down the search results was:

Use the title to eliminate any papers clearly not related to the research focus

Use the abstract and keywords to exclude additional papers not related to the research focus

Read the remaining papers and eliminate any paper that are not related to the research questions

After using the inclusion and exclusion criterion to select applicable papers and studies, a quality assessment was performed on those studies. This quality assessment was another check on the quality of the set of papers that resulted from the initial search.

Each accepted study after using the inclusion and exclusion criterion and removing duplicated studies is assessed for its quality against set of criteria. Some of these criteria were informed by those proposed for the Critical Appraisal Skills Programme (CASP) (in particular, those for assessing the quality of qualitative research) and by principles of good practice for conducting empirical research in software engineering. The criteria covered three main issues pertaining to quality that need to be considered when appraising the studies identified in the review:

- Rigour. Has a thorough and appropriate approach been applied to key research methods in the study?
- Credibility. Are the findings well-presented and meaningful?
- Relevance. How useful are the findings to the software industry and the research community?

Taken together, these criteria provide a measure of the extent to which we could be confident that a particular study's findings could make a valuable contribution to the review. Each of the criteria will be graded on a dichotomous (‘‘yes” or ‘‘no”) scale. The quality assessment criteria are shown in table 3.

VII. EXTRACTION

In the data extraction, data was extracted from each of the primary studies included in this systematic review according to a predefined extraction table as shown in table 4.

VIII. TEST SUITE REDUCTION TECHNIQUE

In this section we demonstrate and explain the main four pre and post test case reduction techniques.

A. CBR (Case-Based Reasoning) Deletion Algorithms Technique (Post-Process)

Table 3 – Quality Assessment Criteria

S. No	Quality Assessment Criteria
1	Is the paper based on research (or is it merely a ‘‘lessons learned” report based on expert opinion)?
2	Is there a clear statement of the aims of the research?
3	Is there an adequate description of the context in which the research was carried out?
4	Was the research design appropriate to address the aims of the research?
5	Was the recruitment strategy appropriate to the aims of the research?
6	Was there a control group with which to compare treatments?
7	Was the data collected in a way that addressed the research issue?
8	Is there a clear statement of findings?
9	Is the study of value for research or practice?

Removing all redundancy test cases is desirable, so many approaches introduced to reduce redundancy test cases. The process of employing artificial intelligent concept in the test case reduction process is considered as an innovated approach in [9].

Case-based reasoning (CBR) is defined by Barry [16] as “one of the Artificial Intelligence-based algorithms, which solve the problems by searching through the case storage for the most similar cases. CBR has to store their solved cases back to their memory or storage in order to learn from their experience.” “Case Base is a collection of cases in CBR, which can be defined as the following: Given a case - base C= {c1... cn}, for c ε C whereas C = CBR, c = case” [16]. For CBR, we discussed three reduction methods that use CBR deletion algorithms: TTCF, TCIF and PCF methods. These methods utilize path-oriented test case generation technique in order to reduce a number of test cases. Path coverage is described by the control flow graph, which is derived from the source-code (program). As example, if we specify S ={s1, s2, s3, s4, s5} to be a set of states in the control flow graph as in figure 1 below, where each state represents a block of code[9].

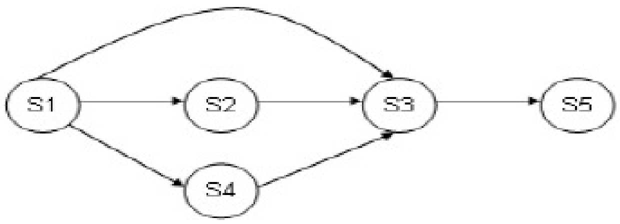


Fig. 1 An Example of Control Flow Graph

From the above figure, we assume that each state can reveal a fault. Thus, an ability to reveal faults of five states is equal to 5. Also, it is assumed that every single transaction must be tested. We will use this example in the three methods of CBR [9].

Let TCn = {s1, s2, …,sn} where TC is a test case and sn is a state or node in the path-oriented graph that is used to be tested. Table 5 summarizes a set of test cases were generated Based on Figure 1.

Table 5 Test Cases

TC1 = {s1, s2}	TC6 = {s1, s4, s3}	TC11 = {s3, s5}
TC2 = {s1, s3}	TC7 = {s1, s2, s3, s5}	TC12 = {s4, s3}
TC3 = {s1, s4}	TC8 = {s1, s4, s3, s5}	TC13 = {s4, s3, s5}
TC4 = {s1, s2, s3}	TC9 = {s2, s3}	
TC5 = {s1, s3, s5}	TC10 = {s2, s3, s5}	

B. Test Case Complexity for Filtering (TCCF)

A complexity of test case is the significant criteria in this proposed method. It measures a number of states included in each test case. Let Cplx(TC) = {High, Medium, Low} where Cplx is a complexity of test case, TC is a test case. The complexity value can be measured as [9]:

- High when a number of states are greater than an average number of states in the test suite.
- Medium when a number of states are equal to an

average number of states in test suites.

- Low when a number of states are less than an average number of states in the test suites.

First, we should produce an auxiliary set from the test suite above. Auxiliary set removes test cases that don't have a direct effect on the ability to reveal faults when it is removed. Therefore, the auxiliary set in our example is as follows [9]:

Auxiliary set = {TC1, TC2, TC3, TC4, TC5, TC6, TC9, TC10, TC11, TC12, TC13}

We can notice that TC7 and TC8 are being removed. Afterward, the method computes a complexity value for all test cases in the above auxiliary set. From figure 1 and the test suite that contain 13 test cases, the average ++

number of states is equal to 3. Therefore, the complexity value for each test case can be computed as follows:

Cplx(TC1) = Low, Cplx(TC2) = Low, Cplx(TC3) =Low, Cplx(TC4) = Medium, Cplx(TC5) = Medium, Cplx(TC6) = Medium, Cplx(TC9) = Low, Cplx(TC10) = Medium, Cplx(TC11) = Low, Cplx(TC12) = Low and Cplx(TC13) = Medium.

Finally, the last step removes test cases with minimum complexity value from the auxiliary set, which they areTC1, TC2, TC3, TC9, TC11 and TC12 .Thus the reduced test suite will be: TC4, TC5, TC6, TC10 and TC13 [9] .

C. Test Case Impact for Filtering (TCIF)

Due to the fact that defining and measuring a quality of software is important and difficult, the impact of inadequate testing must not be ignored. The impact of inadequate testing could be lead to the problem of poor quality, expensive costs and huge time-to-market. In conclusion, software testing engineers require identifying the impact of each test case in order to acknowledge and understand clearly the impact of ignoring some test cases. An impact value is considered here as an impact of test cases in term of the ability to detect faults if those test cases are removed and not be tested [9].

Let Imp(TC) = {High, Medium, Low} where Imp is an impact if a test case is removed, TC is a test case and the impact value can be measured as:

- High if the test case has exposed at least one fault for several times.
- Medium if the test case has exposed faults for only one time.
- Low if the test case has never exposed faults.

The procedure of this method is similar to the previous method. The only different is that this method aims to use an impact value instead of complexity value. The impact value is computed for all test cases in the above auxiliary set, which is {TC1, TC2, TC3, TC4, TC5, TC6, TC9, TC10, TC11, TC12, TC13}. Based on figure 1, the impact value for each test case can be computed as follows:

Imp (TC1) = Low, Imp (TC2) = High, Imp (TC3) =Medium, Imp (TC4) = Low, Imp (TC5) = High, Imp (TC6) = Medium, Imp (TC9) = Low, Imp (TC10) =Low, Imp (TC11) = Low, Imp (TC12) = Low and Imp (TC13) = Low

Finally, test cases with minimum of impact value are removed from the auxiliary set. They are TC1, TC4, TC9, TC10, TC11, TC12 and TC13. Thus the reduced test suite will be: TC2, TC3, TC5, TC6 [9].

D. Path Coverage for Filtering (PCF)

The advantage of path coverage is that it takes responsible for all statements as well as branches across a method. It requires very thorough testing and used as a coverage value in this technique. The coverage value can specify how many nodes that the test case can cover. In other words, the coverage value is an indicator to measure nodes that each test case covers. It means that the higher coverage value is, the more nodes can be contained and covered in the test case.

Let Cov(n) = value, where Cov is a coverage value, value is a number of test cases in each coverage group and n is a coverage relationship.

The first step in this procedure is to identify a coverage set, which can be identified as follows (based on figure 1 above and the set of test cases that derived from it):

Coverage (1) = {TC1}
Coverage (2) = {TC2}
Coverage (3) = {TC3}
Coverage (4) = {TC1, TC4, TC9}
Coverage (5) = {TC2, TC5, TC11}
Coverage (6) = {TC3, TC6, TC12}
Coverage (7) = {TC1, TC4, TC7, TC9, TC10, TC11}
Coverage (8) = {TC3, TC6, TC8, TC11, TC12, TC13}
Coverage (9) = {TC9}
Coverage (10) = {TC9, TC10, TC11}
Coverage (11) = {TC11}
Coverage (12) = {TC12}
Coverage (13) = {TC11, TC12, TC13}

The next step is to calculate a coverage value based on a number of test cases in each coverage group. Therefore, the coverage value can be computed as follows:

Cov (1) = 1, Cov (2) = 1, Cov (3) = 1, Cov (4) = 3, Cov (5) = 3, Cov (6) = 3, Cov (7) = 6, Cov (8) = 6,Cov (9) = 1, Cov (10) = 3, Cov (11) = 1, Cov (12) = 1 and Cov (13) = 3.

The last step removes all test cases with minimum coverage value, in the potential removal set, that they are: TC1, TC2, TC3, TC9, TC11 and TC12. Thus the reduced test suite will be: TC4, TC5, TC6, TC7, TC8, TC10 and TC13 [9].

E. GE & GRE Heuristics and Priority Cost Technique (Post-Process)

GE and GRE heuristics algorithm have been proposed by Chen and Lau [20], Chen et al. defined essential test cases as the opposite of redundant test cases. If a test requirement ri can be satisfied by one and only one test case, the test case is an essential test case. On the other hand, if a test case satisfies only a subset of the test requirements satisfied by another test case, it is a redundant test case [10]. Based on these concepts, the GE and GRE heuristics can be summarized as follows [10]:

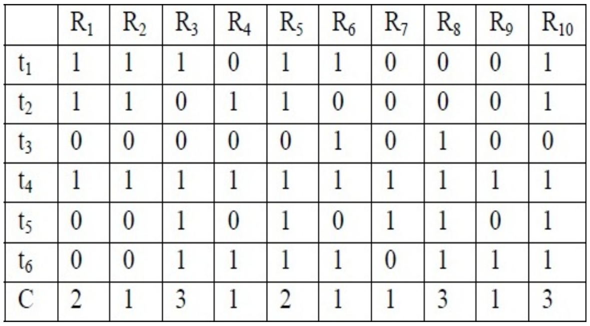
GE heuristic: first select all essential test cases in the test suite; for the remaining test requirements, we use the additional greedy algorithm, i.e. select the test case that satisfies the maximum number of unsatisfied test requirements.

GRE heuristic: first remove all redundant test cases in the test suite, which may make some test cases essential; then perform the GE heuristic on the reduced test suite. A mathematical formula is proposed to reduce the cost of testing by minimizing the size of the test suite using priority based cost. The priority factor will be calculated based on weighted set coverage, the cost of test requirements and test cases [11]. Let us consider the test cases T= {t1, t2, t3, t4, t5, t6} and let requirements of test cases are R= {R1, R2, R3…, R10}.

Requirements according to the test cases (requirements satisfied by each test case) are t1={R1,R2,R3,R5,R6,R10}, t2={R1,R2,R4,R5,R10}, t3={R6,R8}, t4={R1, R2, R3, R4, R5, R6, R7, R8, R9, R10}, t5={R3, R5, R7, R8, R10}, t6={R3, R4, R5, R6, R8, R9, R10}.

After deriving the test cases from the test requirements, each requirement cost (C) is derived and computed from the summation of coverage (such as state coverage, edge coverage or branch coverage), high cost for a requirement means high degree of coverage.

Table 6 Test Cases along with Covered Requirements [11]



	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
t_1	1	1	1	0	1	1	0	0	0	1
t_2	1	1	0	1	1	0	0	0	0	1
t_3	0	0	0	0	0	1	0	1	0	0
t_4	1	1	1	1	1	1	1	1	1	1
t_5	0	0	1	0	1	0	1	1	0	1
t_6	0	0	1	1	1	1	0	1	1	1
C	2	1	3	1	2	1	1	3	1	3

From Table 6 [11], we calculate the cost of each test case, by taking the summation of cost of requirements (that it satisfies) as follows:

Cost (t1) =2+1+3+2+1+3=12

Cost (t2) =2+1+1+2+3=9

Cost (t3) =1+3=4

Cost (t4) =2+1+3+1+2+1+1+3+1+3=18

Cost (t5) =3+2+1+3+3=12

Cost (t6) =3+1+2+1+3+1+3=14.

Next, we checked for unnecessary and redundant test cases, by applying GE and GRE heuristics as mentioned above. If not present, we then calculate the priority factor. We calculate the cardinality of the test cases (requirements satisfied by each test case)[11]:

|req(t1)|=6, |req(t2)|=5, |req(t3)|=2, |req(t4)|=10, |req(t5)|=5, |req(t6)|=7.

The priority of the test case (ti) is then calculated by this formula:

Priority (ti) =Cost (ti) /|req(ti)|

In our example, priorities for the sex test cases are:

Priority (t1) =12/6=2 , Priority (t2) =9/5=1.8, Priority (t3) =4/2=2, Priority (t4) =18/10=1.8, Priority (t5) =12/5=2.4, Priority (t6) =14/7=2

Test cases with lower priority factor will be removed, so t2 and t4 are selected. Thus the reduced test suite will be: t1, t3, t5 and t6 [11].

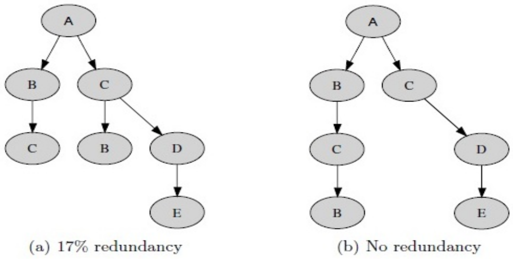
F. Model- Checker Based Technique (Post-Process)

In this technique, we consider test-cases generated with model-checker based methods. A model-checker is a tool originally intended for formal verification. In general, a model-checker takes as input a finite-state model of a system and a temporal logic property and efficiently verifies the complete state space of the model in order to determine whether the property is fulfilled or not [12].

Redundancy is used to describe test-cases that are not needed in order to achieve a certain coverage criterion. As the removal of such test-cases leads to reduced fault detection ability, they are not really redundant in a generic way. In contrast, we say a test-case contains redundancy if part of the test-case does not contribute to the fault detection ability. We are going to identify such redundancy, and describe possibilities to reduce it [12].

Intuitively, identical test-cases are redundant. For any two test-cases t1, t2 such that t1 = t2, any fault that can be detected by t1 is also identified by t2 and vice versa, assuming the test-case execution framework assures identical preconditions for both tests. Similarly, the achieved coverage for any coverage criterion is identical for both t1 and t2.

Clearly, a test-suite does not need both t1 and t2 [12]. The same consideration applies to two test-cases t1 and t2, where t1 is a prefix of t2. t1 is subsumed by t2, therefore any fault that can be detected by t1 is also detected by t2 (but not vice versa). In this case, t1 is redundant and is not needed in any test-suite that contains t2. In model-based testing it is common practice to discard subsumed and identical test-cases at test-case generation time [12]. This kind of redundancy can be illustrated by representing a set of test-cases as a tree. The initial state that all test-cases share is the root-node of this tree. A sub-path is redundant if it occurs in more than one test-case. In the tree representation, any node below the root node that has more than one child node contains redundancy. If there are different initial states, then there is one tree for each initial state. The depth of the tree equals the length of the longest test-case in TS. Children(x) denotes the set of child nodes of node x. Consider a test-suite consisting of three test-cases (letters represent distinct states): ”A-B-C”, ”A-C-B ”, ”A-C-D-E”. The execution tree representation of these test-cases can be seen in Figure 2(a) [12]. The rightmost C-state has two children, therefore the sub-path A-C is contained in two test-cases; it is redundant.



(a) 17% redundancy (b) No redundancy

Fig. 2: Simple test-suite with redundancy represented as execution tree.

The execution tree can be used to measure the redundancy R of test-suite TS based on the following relation:

$$R(TS) = 1/(n-1) \cdot \sum_{x \in children(root(TS))} R(x) \qquad (1)$$

The redundancy of the tree is the ratio of the sum of the redundancy values R for the children of the root-node and the number of arcs in the tree (n − 1, with n nodes).

The redundancy value R is defined recursively as following relation [12]:

$$R(x) = \begin{cases} (|children(x) - 1|) + \sum_{c \in children(x)} R(c) & \text{if } children(x) \neq \{\} \\ 0 & \text{if } children(x) = \{\} \end{cases} \qquad (2)$$

The example test-suite depicted as tree in Figure 2(a) has a total of 7 nodes, where one node besides the root node has more than one child, which is the node c. Therefore, the redundancy of this tree (based on relations 1 and 2) equals:

$$R = 1/(7-1) \cdot \sum_{x \in children(root(TS))} R(x)$$

$$R = 1/6 \; . \; (0 + (1+0)) = 1/6 = 17\%$$

A test-suite contains no redundancy if for each initial state (root node) there are no test-cases with common prefixes, e.g., if there is only one test-case per initial-state. Figure 2(b) illustrates the result of an optimization applied to the Figure 2(a) [12] in order to remove redundancy. The test-cases A-C-B and A-C-D-E have the common prefix A-C, and there is a test-case ending in C, which is A-B-C. Therefore the postfix B of A-C-B is appended to A-B-C, resulting in A-B-C-B. Thus test suite with the three test cases is reduced to become test suite with two test cases after removing the redundancy [12].

G. Base Choice Coverage Criterion Technique (Pre-Process)

The input domain to any program contains all the possible inputs to that program. In equivalence partitioning technique, the domain for each input is partitioned into regions (partitions), and each partition defines a set of blocks that must be pair wise disjoint (no overlap) and covers the domain of each partition (complete), as we can see in figure 3 [15].

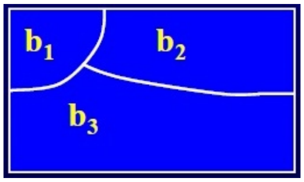


Fig. 3 three blocks for a partition which are disjoint and complete

An important question would be: “How should we consider multiple partitions at the same time?” This is the same as asking “What combination of blocks should we choose values from?” The most obvious choice is to choose all combinations. However, using all combinations will be impractical when more than 2 or 3 partitions are defined [15]. For example, if we have three partitions with blocks [A, B], [1, 2, 3] and [x, y]. Table 7shows the twelve test cases are needed for all combinations coverage.

Table 7 Combinations Coverage Test Cases

t1 : (A, 1, x)	t5 : (A, 3, x)	t9 : (B, 2, x)
t2 : (A, 1, y)	t6 : (A, 3, y)	t10 : (B, 2, y)
t3 : (A, 2, x)	t7 : (B, 1, x)	t11 : (B, 3, x)
t4 : (A, 2, y)	t8 : (B, 1, y)	t12 : (B, 3, y)

Ammann and Offutt [13] advocated base choice coverage criterion as the minimum adequate criterion. They argued that each system has a normal mode of operation and that normal mode corresponds to a particular choice in each category (partition). This particular choice (block) is called as base choice. Thus base – choice - coverage criterion requires that each choice in a category be tested by combining it with the base choice for all other categories. This causes each non-base choice to be used at least once, and the base choices to be used several times [14].

We simply ask: What is the most "important" block for each partition in our domain? This block is called the "base choice" [15]. For our example above, we suppose that base choice block in partition [A, B] is A, in partition [1, 2, 3] is 1 and in partition [x, y] is x. Then a base choice test case and additional test cases would be like the following [15]:

T1 : (A, 1, x) which is called the base test	t2 : (B, 1, x)	t3 : (A, 2, x)	t4 : (A, 3, x)	t5 : (A, 1, y)

As we can see, in base choice coverage criterion the number of test cases are reduced compared with all combinations coverage criterion. This is because of choosing a base choice block for each partition we have. Which blocks are chosen for the base choices becomes a crucial step in test design that can greatly impact the resulting test [15].

Equivalence class	Valid value	Invalid value	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9
Foreground color	Red	Lavendel is a beauti	Red	Blue	Black	White	Lavend	Purple	Red	Red	Red
	Blue	Purple									
	Black										
	White										
Background color	Yellow	White	Yellow	Brown	Brown	Brown	Yellow	Yellow	White	Black	Yellow
	Brown	Black									
Outlining	Left	Center	Left	Right	Right	Right	Left	Left	Left	Left	Center
	Right										

Fig.4 Equivalence partitioning organizer tool example

Figure 4 represents an example of equivalence partitioning for a specific inputs, organized using the equivalence partitioning organizer [33] (written by Martin Keesen –version 0.5). The organizer allows us to create partitions with their valid and invalid values (blocks). In the above example, there are three partitions: Foreground color, Background color and Outlining partitions. Each has their own valid and invalid values. From the edit menu, we choose: Auto create test cases, and then the test cases above will be generated automatically based on a coverage criteria called: Each choice coverage, which requires that: one value from each block for each partition must be used in at least one test case[15]. So we notice that the four test cases (TC1-TC4) have covered all valid blocks each at least one and the last five test cases form (TC5-TC9), represent possible combination between one invalid block with two other valid blocks from the different partitions.

H. *Pros and Cons of Test Case Reduction Techniques*

There are many research challenges and gaps in the test case reduction area. Those challenges could inspire interested researchers to further inspect this area to use most effective reduction techniques.. However, the research issues that motivated this study are: the too many redundancy test cases after reduction process, a decrease of test cases ability to reveal faults and the uncontrollable grow of test cases [9]. Table 8 summarizes the advantages and limitations of the aforementioned test suite reduction techniques:

The tester is likely to dramatically increase his or her understanding of the software by deriving the FSMs, and then deriving tests from them. Some Simple and straightforward suggestions are exist for generating FSMs from code, Like using the software structure, modeling state variables (global and class) or using the implicit or explicit specifications [15]. Next we present a tool that helps us to write or draw FSMs and easily generate tests automatically.

Table 8 Pros and Cons of the four Reduction Techniques

Technique / Algorithm	Advantages	Limitations
CBR algorithms	- Preserving capability to detect faults after reduction (especially TCCF and TCF) [9]. - Removing the redundancy and unnecessary test cases[11] - Controlling the growth of test cases [11].	- Require a lot of time.(specially TCCF and TCF) [9]. - The path coverage may be not an effective coverage factor for a huge system that contains million lines of code. This is because it requires an exhaustive time and cost for identifying coverage from a huge amount of codes [9].
GE & GRE Heuristics and priority cost technique	- Construction of optimal representative set [11]. - Reduce the redundant and unnecessary test cases [11].	- T he NP-complete problem [11].(that is no fast solution is known)
Model-Checker	- A convenient tool for optimization purposes and removing redundancy, especially if it is already used for test-	- Not an effective for a huge system that contains million lines of code [9]. Because this will be costly

	case generation in the first place [12]. - Quality of the resulting test-suites does not suffer with regard to test coverage or fault detection ability [12].	and time consuming.
Base choice criteria (equivalence partitioning)	- Fairly easy to get started, because it can be applied with no automation and very little training [15]. - Simple to tune the technique to get more or fewer tests [15].	- Quality of the resulting test-suite may suffer or be not efficient in revealing defects, because choosing base choices is crucial step that depends on the tester.

REFERENCES

[1] Jovanović, I., Software Testing Methods And Techniques, May 26,2008.

[2] Guide To The Software Engineering Body of Knowledge, Swebok: A Project of the IEEE Computer Society Professional Practices Committee, 2004.

[3] Miller, E., Introduction to Software Testing Technology, Tutorial: Software Testing & Validation Techniques, Second Edition, IEEE Catalog No. EHO 180-0, 1981, pp. 4-16.

[4] Marciniak, J., Encyclopedia of Software Engineering, Vol. 2, New York, NY: Wiley, 1994, pp.1327-1358.

[5] Khan, M., Different Forms of Software Testing Techniques For Finding Errors, IJCSI International Journal Of Computer Science Issues, Vol. 7, No. 3, No 1, May 2010.

[6] Abhijit, A., Sawant1, P. H. Bari2 & P. M. Chawan3, Software Testing Techniques and Strategies, Vol. 2, No 3, 2012, pp.980-986.

[7] Beizer, B., Software Testing Techniques, Van Nostrand Reinhold, 2nd Edition, 1990.

[8] Mahapatra, R., & J. Singh, Improving the Effectiveness of Software Testing Through Test Case Reduction, World Academy of Science, Engineering and Technology, 2008.

[9] Roongruangsuwan, S. & Daengdej, J., Test Case Reduction Methods by Using CBR,Autonomous System Research Laboratory Faculty of Science and Technology Assumption University, Thailand.

[10] S. Yoo, M. Harman, Regression Testing Minimisation, Selection and Prioritisation: A Survey" Softw. Test, Verif. Reliability, 2007, (DOI: 10.1002/000)

[11] Rout, J. Et al, An Effective Test Suite Reduction Using Priority Cost Technique, International Journal of Computer Science & Engineering Technology, (IJCSET), 2011.

[12] Fraser, G. & F. Wotawa, Redundancy Based Test-Suite Reduction, In Matthew B. Dwyer & Ant´Onia Lopes, Editors, FASE, Vol. 4422 of Lecture Notes In Computer Science, 2007, pp. 291–305.

[13] Ammann, P. & J. Ofutt, Using Formal Methods to Derive Test Frames In Category- Partition Testing, Proceedings of the Ninth Annual Conference On Computer Assurance (COMPASS 94), 1994, pp. 69-80.

[14] Ghani, K., Searching For Test Data, 2009.

[15] Ammann, P & J. Ofutt, Introduction To Software Testing Cambridge, UK, ISBN 0- 52188-038-6, 2008.

[16] Boehm, B., A Spiral Model Of Software Development and Enhancement, TRW Defense Systems Group, 1998.

[17] Rothermel, G., M. Harrold, J. Ostrin & C. Hong, An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites, Proceedings of IEEE International Test Conference on Software Maintenance (ITCSM'98), Washington D.C.,1998, pp. 34-43.

[18] Mcmaster, S. & A. Memon, Call Stack Coverage For Test Suite Reduction, Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005, pp. 539-548, Budapest, Hungary.

[19] Sampath, S., S. Sprenkle, E. Gibson & L. Pollock, Web Application Testing With Customized Test Requirements – An Experimental Comparison Study, 17th International Symposium On Software Reliability Engineering (ISSRE'06), 2006.

[20] Chen, T & M. Lau, A New Heuristic For Test Suite Reduction, Information And Software Technology, Vol. 40, No. 5-6, 1998, pp. 347–354.

[21] Horgan J., S. London, ATAC: A Data Flow Coverage Testing Tool For C. Proceedings of The Symposium on Assessment of Quality Software Development Tools, IEEE Computer Society Press, 1992, pp. 2–10.

[22] Horgan JR, S. London , Data Flow Coverage and the C Language. Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4), ACM Press, 1991,pp. 87–97.

[23] Bertolino, M. , Using Spanning Sets For Coverage Testing, IEEE Transactions on Software Engineering, Vol. 29, No. 11, 2003, pp. 974–984.

[24] Black, P.E.: Modeling And Marshaling: Making Tests From Model Checker Coun-Terexamples. In: Proc. Of The 19th Digital Avionics Systems Conference, (2000).

[25] Heimdahl, M. & G. Devaraj, Test-Suite Reduction For Model Based Tests: Effects on Test Quality And Implications For Testing. In: ASE, IEEE Computer Society, 2004, pp. 176–185.

[26] Arnicane, V., Complexity of Equivalence Class And Boundary Value Testing Methods, International Journal of Computer Science and Information Technology, Vol. 751, 2009, pp. 80-101.

[27] Demillo, R., W. Mccracken, R. Martin, & J. Passafiume, Software Testing and Evaluation, Benjamin/Cummings Publishing Company, Menlo Park, CA, 1987.

[28] White, L. & E. Cohen, A Domain Strategy For Computer Program Testing, IEEE Transactions On Software Engineering SE-6, 1980, pp. 247-257.

[29] Jeng B., Weyuker E. J., A Simplified Domain-Testing Strategy. ACM Trans. Softw. Eng. Methodol. Vol. 3, No. 3, 1994, pp. 254– 270.

[30] Zeil, S. & F. Afifi, White L. J. Detection of Linear Errors Via Domain Testing, ACM Trans. Softw. Eng. Methodol. Vol.1, No.4, 1992, pp. 422–451.

[31] Utting, M., G. Perrone, J. Winchester, S. Thompson, R. Yang & P. Douangsavanh, The Modeljunit Model-Based Testing Tool - Department of Computer Science, The University of Waikato, New Zealand.

[32] http://Sourceforge.Net/Projects/ Equivalencepart/, 2004, pp. 247–257.

Dr. Mohammed Akour is an Assistant Professor in the Department of Computer Information System at Yarmouk University (YU). He got his Bachelor (2006) and Master (2008) degree from Yarmouk University in Computer Information System with Honor. He joined YU as a Lecturer in August 2008 after graduating with his master in Computer Information System. In August 2009, He left YU to pursue his PhD in Software Engineering at North Dakota State University (NDSU). He joined YU again in April 2012 after graduating with his PhD in Software Engineering from NDSU with Honor.

Dr. Iyad Alazzam is an assistant professor in the department of computer information systems at Yarmouk University in Jordan, he has received his Ph.D degree in software engineering from NDSU (USA). His master from LMU (UK) in electronic Commerce and his B.Sc in computer science and information systems from Jordan University of Science and Technology in Jordan. His research interests lays in software engineering and software testing.

Dr. Feras Hanandeh is an Associate Professor in the Faculty of Prince Al-Hussein Bin Abdallah II for Information Technology. His research interest is in Constraints Integrity Maintenance for Parallel Databases, Distributed Databases, Artificial Intelligence, and Grid Computing.

Dr. Iman Akour is an associate Professor of Management Information Systems at Sharjah University. Got her PhD in Business Administration, (2006), from Louisiana Tech University, USA. Specialization: Information System & Quantitative Analysis. Her master was in Business Administration, (1995), Grambling State University, USA. Major: Computer Information Systems. She supervised several graduate dissertations in USA and was a committee member for graduate student's thesis from other countries.