ISSN:0975-9646

# Approaches on Detecting Domain Errors – A Survey

Dr. Anna Saro Vijendran*1, N.R.Suganya#2

*1*Director, Dept of Computer Applications, SNR Sons College*
*Coimbatore, Tamilnadu, India*

#2*Research Scholar, Karpagam University*
*Coimbatore, Tamilnadu, India*

*Abstract* - **The past decades of research in software testing has foreseen the development of n number of techniques for assessing the domain correctness of software units. A method of programming and debugging causes many types of errors. Types of errors would have some classifications based on the nature of the program. One of those is the occurrence of domain errors in programs. The major challenge in this area is to generate a set of test cases automatically and test it with various techniques. Main goal of this survey is to allow the software testers and developers to become aware of where the error origins and how it would be tested? The following steps are elucidated in detail of how a program should be tested - by test case generation, by test case tools and by test case algorithms with the presence of some correct behavior in programs. This paper presents a short survey of past decades on how the errors have been handled by the testers.**

*Keywords*— **Domain Errors, Test Case Generation, Tools, Algorithms, Approaches and Methods.**

## I. INTRODUCTION

Testing is done in an intention of finding errors. "A program is said to exhibit a domain error when incorrect output is generated due to executing the wrong path through the program". Simple example for a domain error is when a floating point is rounded off to an integer. A domain error occurs when an input value origins the program to execute the wrong path. A program is said to have a domain error if it erroneously performs input cataloguing. Domain errors are being tested by a testing technique known as domain testing. Domain testing is one of the major categories of software testing. The core of domain testing is that to partition a domain into sub domains (equivalence classes) and then select some representatives of each sub domain for our tests. A domain error arises from incorrect implementation of designed domains and the way of testing provides finding of errors in the numeric expressions affecting the flow of control through the program. There are many testing techniques which deal with the detection of domain errors in some functions using only a single additional test point. This paper provides the different test case generation tools and algorithms which will help the testers to prevent domain errors in their projects.

## II. REVIEW OF LITERATURES

In the olden days the domain errors where identified based on the feasibility of data gathering, error density, means of error detection, reason and nature of code change, cases were change not required, effort to diagnose, effort to correct and the efficiency of hand processing Vs computer testing says M.L.Shooman & M.I.Bolsky [1]. Based on these techniques the types of errors, their nature and their frequency with the effort to diagnose and the way to correct them can be described. Whatever the dimensions may be (linear/non-linear) with dimensions either two or three the errors can be detected.

A frequency domain approach was adopted in 1997 to tackle the problem of validating uncertainty models described by linear fractional transforms. This problem amounts to verifying the consistency of certain given mathematical models to experimental information obtained from a physical plant, using either input–output, frequency-domain measurements or frequency samples of the plant. Linear fractional models with both unstructured and structured uncertainties are considered. The problem is resolved in the former case and solved approximately in the latter. Both results lead to tests that are readily computable via convex optimization methods and can be implemented using standard algorithms. In comparison to previously available algorithms based on time-domain information, the main advantage of these tests is that they have a considerably lower level of computational complexity. It is shown that the validation problem reduces to one of Nevanlinna–Pick boundary interpolation, and it can be solved by computing independently a sequence of convex programs of a lower dimension, each of which corresponds to only one frequency sample. Many mathematical formulations and validated examples are given for this domain approach. [2]

Systems with the length-based Church-Rosser property have a highly efficient method of reduction of a sting to a canonical form. It shows how to construct, for any such system, an automaton with two pushdown stores that can reduce any string over the alphabet to its canonical form in time that is linear in the length of the string. It was assume that we have a mixed system (C, *E, R)* with the length-based Church-Rosser property. To execute the first reduction step of a given string, we must find a factor of that string that is the left member of a rule of *R;* such a factor let us call a "handle." There may be several handles in the string, so we must decide both how we should begin our search for handles and which handle should be the first to be rewritten. Eventually we shall come to a sting without a handle, at which point the reduction is complete: the final string is an irreducible equivalent of the original string. And, since the system has the Church-Rosser property, it is the only irreducible equivalent string.

Suppose in a given step of the procedure that we have reduced $w_l x\ w_2$ to $w_l\ y\ w_2$, where (x, y) □ *R,* and where wt is long. In order to find the leftmost handle in $w_l\ y\ w_2$, we do not have to begin our search at the left end of $w_l$. We can be sure from what has happened so far that $w_1$ has no handle. (We omit the proof of this fact, which is by mathematical induction on the number of reduction steps that have taken place.) More precisely, let h be the length of the longest left side of a rule of *R* minus 1. If $|w_1| > h$ then, taking $w_1 = w_{12} w_{13}$ where $|w_{13}| = h$, we can confine our search to $w_{13}$ y $w_2$, ignoring $w_{12}$ completely for this step. If $|w_1| < h$ then, of course, we must begin our search at the left end of $w_1$. This completes our description of the algorithm, from which it can be proved that it will always result in the unique irreducible string equivalent to the original, provided that the system has the Church-Rosser property. Everything that has been said so far about the algorithm holds even if the Church-Rosser property is not the length based property. However, the analysis that follows, showing that it is a linear-time algorithm, requires the length-based property. If the system is not Church-Rosser at all, an equivalent irreducible string will be found, but there is no guarantee that it will be unique or have minimal length. In order to analyze the algorithm it is convenient to modify the notion of step. Let us stipulate that the algorithm begins at time 0 with a pointer at the leftmost character of the input string. Thereafter, the string will be modified and the pointer will be moved.

$$k_l + k_2 < k_1 + (l + h)\, g < (2+h) g.$$

And so the author is able to conclude that the computation time for the algorithm is bounded by a linear function of g. This algorithm would be easily implemented as a computer program, which, if care is taken in the writing, runs in linear time. [3]

For dataflow and domain testing, the control structure of the program being tested is represented by a directed graph (digraph), called the control flow graph or flow graph. A digraph G = (V, E ) consists of a set of nodes V, and a set of edges E. A node $v_i$ represents a basic block (a single-entry single-exit sequence of code always executed together), and an edge $(v_i , v_j)$. Represents possible transfers of control from node i to node j. If $(v_i , v_j)$ □ E, node $v_i$ is a predecessor of $v_j$ and $v_j$ is a successor of $v_i$. Without loss of generality we assume that G has a unique node with no predecessor, the source s, and a unique node with no successor, the sink t. A path from $v_i$ to $v_j$ is an alternating sequence of nodes and edges starting with $v_i$ and ending with $v_j$. An execution of the program corresponds to a complete path from the source s to the sink t. While, in a program, predicates are expressed by the program variables, it is useful to represent them in terms of the input variables. This is done by a technique called symbolic execution which assigns a symbolic name to each input variable and executes the path through the program. A predicate in this path, after being symbolic executed, will generate a result called a predicate interpretation, in which each variable appearing in the predicate is replaced in terms of input variables in symbolic forms. The path domain with respect to a path p, denoted dom(p), is defined by the path condition associated with p. If the path condition is consistent, then dom(p) is not empty and the path is feasible; otherwise dom(p) is empty, and the path is infeasible. For a deterministic program, the intersection of dom(p) and dom(q) will be empty if p and q are different since each input traverses exactly one path.

Fault analysis shows that the effectiveness of a testing strategy depends on the satisfaction of both path-discrimination and data-discrimination conditions as discussed earlier. Since few of the existing strategies meet both conditions, in this article we have proposed an integration of two different testing techniques in order to make use of each one's strengths. Data flow testing traces the behavior of a variable definition (path-discrimination), but is not effective in the detection of certain faults. Domain testing is effective in fault detection within a path domain (data-discrimination), but lacks a criterion to guide the selection of testing paths. Integration inherits their strengths but not their deficiencies. It meets both path-discrimination and data-discrimination conditions. By a pair of carefully selected test cases, its correctness will then be reflected as different branches taken by the path. That is, a wrong path will be traversed if a fault does exist, which is of course easier to judge and be found. Evaluation of the fault-detection ability of a testing strategy, however, is very difficult. More indicative experiments will have to wait until a tool that implements this testing strategy is created so that many programs can be executed automatically. This approach, however, is a new breed which shows that the two techniques need not be treated separately (i.e., it is both structural and fault based). Since it has been widely accepted that none of the testing strategies is without deficiencies, and usually one's weak points are another's strong points, it is believed that the combination of different testing strategies to gather together their strengths is a promising research direction. The integration of data flow and domain testing strategy proposed in this paper illustrates such a possibility. [4]

The next approach introduces test classes and a test class framework for generating test cases from Z specifications. We define a test class using object-oriented concept in test framework instead of Phil Stock’s test template. This test framework for Z specifications uniformly defines the test data and oracles in a test class that also contains the information of before states and after states for an operation. Thus, the derivation and construction of test case and test sequence information can be unified in a test framework. It presents an example to demonstrate how to generate test cases using the test framework. To support the framework, we have designed and implemented a test case generation system, TCGS, and its functions.

The Author has used a structured approach to build a hierarchy of test class. Our test class hierarchy is similar to Phil Stock's test template hierarchy. We have designed and implemented a test case generation system TCGS which is a subsystem of our Z User *Studio,* a Z notation support system. TCGS produces test cases for an operation from Z specification. The generated test cases are saved in a file which takes the name of operation schema as file name and .tst as extension name. The whole process of generating test cases is defined in Z schema. It runs under Windows98. The user can browse the contents of a test class by clicking the test class. For sake of space, we can not describe details of the design and implementation of TCGS. We only show the part of execution interface. When starting testing, TCGS shows dialog box ***start test,*** the user then select the name of operation schema under test.

The author has defined a test class framework using object-oriented concept. The benefit of this test framework for Z specifications is that the test data and oracles are uniformly defined in a test class which also contains the information of before states and after states for an operation. Thus, the derivation and construction of test cases and test sequence information can be unified in a test framework. [5]

Classification tree is one of the methods of generating test cases from specification given by the author. It partitions the input domain into a number of classifications. A classification tree is created to depict the relationship among the classifications. Test cases were then derived from this tree. However classification trees have a number of shortcomings. The expressive power is limited by the tree structure. It also relies much on human decision in selecting test cases. This paper has introduced an alternative approach to the generation of test cases. It also defines classifications and classes formally. Then, it analyzes the relations among classes and classifications and expresses these relations in vectors (class vectors). Test cases are then derived from the Cartesian product of these partitions of vectors. The expressive power of vectors is better than tree structure and hence can be applied in some systems which test cases are complicated. Furthermore, more information obtained in the specifications were utilized. Hence the amount of human decision is minimized.

The classification tree method (CT) is a black box testing technique. The fundamental concept of CT is to divide the input domain of a program into disjoint sets of classes, and to form test cases by choosing from these classes according to certain criteria. A CT consists of two major components namely classifications and classes. Classifications are defined as aspects of viewing the input domain of the program to be tested. Classes are defined as disjoint subsets of values of classifications. This method consists of four steps which are summarized as follows.

- Identify all the classifications and their classes from the specifications.
- Construct the CT with the classifications and classes.
- Construct the combination table from the classification tree.
- Select all the feasible combinations of classes from the combination table, thus each selected combination of classes constitute one test case. This is the step where manual decisions are made.

There is certain weakness with CT. First, although CT can help to structure the classifications and classes, it can lead to many invalid test cases which require manual decisions to filter them out. Secondly, the real relations should be those relations between classes. For example, in, CT has to be constructed with reference to the relations between classes. The classifications play little role in the test case generation. Thirdly, not all relations between classes can be denoted by a tree structure. For example, the relations between Card Class, Credit Limit, Purchase Balance and Cumulative Balance were not recorded in the classification tree. Consequently, manual decisions are required to choose those legitimate test cases. A number of redundancies of classifications will be required when there are complex relations between classifications. For example, in our case study, it is required to duplicate the classification air-ticket type. This is because the relations between the classification air-ticket type and each of the classes CX and DG are different; the classification has to be duplicated in the tree. These duplications will increase the complexity of the classification tree. The increase in complexity will increase the chance of making mistakes. There are also problems with the expressive power of classification tree. For example, according to the specifications, the classification purchase balance has to be partitioned into four classes namely Al, A2, A3 and A4. However, the classification tree cannot express the information that there is no relation between A4 and the classification credit limit. This has to be determined by manual decisions during test case generation. CT has no formal semantics which may cause ambiguity in performing analysis.

The Classification Tree method of generating test cases from specification and found that there are a number of shortcomings. Two of these, namely the inability to capture certain complex relations among the classes and the need of manual filtering of test cases, are considered very undesirable. One major cause is due to the tree structure that is unable to utilize all information in the specification and to express the complexity of the specification. To overcome this, the author has proposed an alternative approach. It uses more general structures, relations and vectors, to capture the information in the specification and to represent the test cases. It formally defined a number of fundamental concepts about legitimate test cases and developed a procedure of generating test cases using these definitions. On the whole, this approach was more algebraic. One of the reasons of this approach is that it aims at developing automated or mechanical tools to support the generation of test cases. On the other hand, it has been working on a more visual version along the similar approach. In that graphs are used to express the relations among the classes and classifications. [6]

The significant expansion of autonomous control and information processing capabilities in the coming generation of mission software systems results in a qualitatively larger space of behaviors that needs to be “covered’ during testing, not only at the system level but also at subsystem and unit levels. A major challenge in this area is to automatically generate a relatively small set of test cases that, collectively, guarantees a selected degree of coverage of the behavior space. The below part describes an algorithm for a parametric test case generation tool that applies a combinatorial design approach to the selection of candidate test cases. Evaluation of this algorithm on test parameters from the Deep Space One mission reveals a valuable reduction in the number of test cases, when compared to an earlier home-brewed generator.

Both TCG and AETG use the greedy approach, which assumes that there is no need to backtrack to find potentially better solution. This simplifies the algorithm and reduces the computation time, yet the result is proven to be close to optimal. The TCG algorithm differs from AETG’s mainly in that the former uses a deterministic method while the latter is using random selection. In the AETG algorithm, each partial test case is determined by first generating $M$ different candidate test cases and then choosing one that covers the most new pairs, where $M$ is selected to be 50 for best result. We found that when there is more than one “greatest” or “least” number of something when carrying out the algorithm, selecting one with a fixed order (such as top first) will end up with much less than optimal result. The largest number of values in the set defines the lower bound of the number of test cases to be generated in a 1-way coverage (i.e. when every parameter is independent of each other), because each value of that parameter is required to appear at least once. In the case of n-way coverage, meaning every combination of *12* parameters has to have .all possible values in’ the set of selected test cases; we use the first 12 parameters or the *II* largest-sized parameters, to find the lower bound of test cases to be generated. The TCG algorithm starts to build up test cases from values selected for these first *11* parameters.

At times a selected test case covers more new pairs than a previous selected test case. This suggests that we did get a sub-optimal test case with the “greedy” approach. To improve this we could take a “non-greedy‘ approach and back tract one or a few steps. But since this approach will complicate the algorithm and will increase computation steps, it is not clear if the extra costs on the algorithm development and the computation time are worth while. If so, we may elect to switch from “greedy” to “back-tracking’’ when, and only when we run towards the end of the algorithm. It will be interesting to learn how much we can improve using this approach. The MDS TCG tool is being implemented in Java. While the core part of the algorithm has been completed and tested, its other features are still limited at this writing. There are useful features currently being implemented, including the following:

- Seeds.
- N-way coverage.
- Constraints.

The TCG algorithm illustrates the value of a combinatorial design approach in parameter based test case generation. Nearly identical results with TCG and AETG confirm that this algorithm is on a par with the commercial state of the art. By implementing this algorithm as a reusable software component, it can then be embedded in a variety of test harnesses. This helps a project’s verification effort apply a consistent, disciplined approach to achieving a selected degree of test coverage with a near-minimal number of test cases. [7]

Errors are classified in two main categories, Non effective and Effective errors. The Non-effective errors correspond to errors, which were either latent or overwritten. Latent errors indicate that the injected fault had no effect on the program execution, but the observable state of the CPU differed from the fault-free state when the program finished. Overwritten errors indicate that the injected fault was overwritten without causing any other effect on the system. [8]

The next approach introduces DIDUCE (Dynamic Invariant Detection ∪ Checking Engine)., a practical and effective tool that aids programmers in detecting complex program errors and identifying their root causes. By incrementing a program and observing its behavior as it runs, DIDUCE dynamically formulates hypotheses of invariants obeyed by the program. DIDUCE hypothesizes the strictest invariants at the beginning, and gradually relaxes the hypothesis as violations are detected to allow for new behavior. The violations reported help users to catch software bugs as soon as they occur. They also give programmers new visibility into the behavior of the programs such as identifying rare comer cases in the program logic or even locating hidden errors that corrupt the program's results. The author has implemented the DIDUCE system for Java programs and applied it to four programs of significant size and complexity. DIDUCE succeeded in identifying the root causes of programming errors in each of the programs quickly and automatically. In particular, DIDUCE is effective in isolating a timing-dependent bug in a released JSSE (Java Secure Socket Extension) library, which would have taken experienced programmer days to find. Our experience suggests that detecting and checking program invariants dynamically is a simple and effective methodology for debugging many different kinds of program errors across a wide variety of application domains.

DIDUCE was especially helpful in pinpointing late-stage bugs that occur after many test cases are already running. Late-stage bugs are usually the hardest to find and take the longest to analyse. Experimentation with four real-life applications suggests that DIDUCE is effective in detecting hidden errors and finding the root causes of complex programming errors. It can find bugs that result from algorithmic errors in handling corner cases, errors in inputs, and developers’ misconceptions of the APIs. It helps programmers locate bugs in unfamiliar code and, sometimes even in codes that have not been instrumented. Furthermore, no up-front investment is required; users start using DIDUCE only when they are confronted with a bug, or the possibility of one. While we used only the simple, default invariants in our experiments, users can tailor DIDUCE to check for more complex invariants to suit the specific application. [9]

CPM is a specification-based testing technique developed by Ostrand and Balcer. It helps software testers create test cases by refining the functional specification of a program into test specifications. It identifies the elements that influence the functions of the program and generates test cases by methodically varying these elements over all values of interest. The method consists of the following steps:

- Decompose the functional specification into functional units that can be tested independently.
- Identify the parameters (the explicit inputs to a functional unit) and environment conditions (the state of the system at the time of execution) that affect the execution behavior of the function.
- Find categories (major properties or characteristics) of information that characterize each parameter and environment condition.
- Partition each category into choices, which include all the different kinds of values that are possible for that category.
- Determine the constraints among the choices of different categories. For example, one choice may require that another is absent or has a particular value.
- Write the test specification (which is a list of categories, choices, and constraints in a predefined format) using the test specification language TSL.

- Use a generator to produce test frames from the test specification. Each generated test frame is a set of choices such that each category contributes no more than one choice.
- For each generated test frame, create a test case by selecting a single element from each choice in that test frame.

They have developed a choice relation framework for supporting category-partition test case generation. The major merits of the framework are:

- We capture the constraints among choices in a rigorous and systematic manner via the introduction of various relations.
- We improve on the effectiveness and efficiency of complete test frame construction by means of consistency checks and automatic deductions of relations.
- We provide a means of removing only the incorrectly defined relations and any related ones, thereby saving the effort of repeating the entire construction process for the choice relation table.
- We provide a direct way to control the maximum number of generated test frames. We enable the software tester to specify the relative priorities for choices that are used for the subsequent formation of complete test frames.

The author has applied his approach to real-life situations and reported on the effectiveness of consistency checks and automatic deductions of choice relations. [10]

Because of the complexity of testing at the system level, it is appropriate to use more than one mode to represent system behavior. The approach taken in this work is to establish two levels of modeling to be used in the generation of test cases. *Behavior modeling* is established to capture high-level sequential behaviors required of the system. *Data modeling* is established to manage the concrete test data values for each scenario in order to generate executable test vectors. This approach is motivated by our experience with the limitations using single model techniques at the system level. The work is also motivated by approaches using formal specifications, especially, in which several different models are used to support test case generation.

A test scenario is a high level abstraction of system behavior that describes an important sequence of actions to test, but it is insufficient to make a test vector that is executable in a test harness because it lacks concrete test data values. Under our approach, a data model is created to manage the SUT’s test data. Models are useful in the generation of test cases to the extent that they support “good” testing ideas. Models that restrict a tester’s vision as to what can or cannot be tested, or models that obscure good testing ideas in millions of generated test cases are less than useful. We have introduced an approach to test case generation that combines behavior and data modeling. A data model is created at the level of sophistication warranted by the importance of each test scenario. The data model allows the tester to manage the SUT’s test data and to address the expected outcomes of test cases using IO relationships. A positive outcome is that large, fully automated test suites are generated and executed. In addition, two cases studies indicate that a larger number of the required test cases were generated using the combined approach than were generated when using data modeling alone. While the approach presented will not be applicable in all testing situations, we believe it should be considered as way to improve test case generation. [11]

The author has presented a novel approach to automatically generate *ON-OFF* test points for character string predicate borders associated with program paths, and develop a corresponding test data generator. Instead of using symbolic execution or program instrumentation, it constructs a slice with respect to a predicate on a path via program slicing techniques. The current values of variables in the predicate are calculated by executing the slice, thus avoiding the problems found in symbolic execution and the costly and time-consuming jobs for designing proper instrumentation statements. Each element of variables in a character string predicate is determined in turn by performing function minimization so that the *ON-OFF* test points for the corresponding predicate border are automatically generated.

All recent domain testing strategies have been limited to programs in which character string predicates are not taken into consideration. The same weakness is found in many currently available test data generation system. In his paper, he has presented a novel approach to automatically generate *ON-OFF* test points for character string predicate borders associated with program paths, and develop a corresponding test data generator by Jeng simplified domain testing strategy. Symbolic execution or program instrumentation is not involved in the system. Instead, a predicate slice is constructed to calculate the current values of variables in the predicate, avoiding the problems found in symbolic execution and the cost of designing proper instrumentation codes. [12]

A classification of program errors with strong intuitive appeal is the division into domain errors and computation errors, in which errors usually arise from the predicate faults in conditional statements or from assignment faults in a program. A domain error can be manifested by a shift or a tilt in some segment of the path domain boundary and therefore incorrect output is generated due to executing a wrong path through the program. Domain testing is applicable whenever the input domain is divided into sub domains by the programs decision statements. A test point is a set of values of all the input variables in which one value is bound to one variable. In this approach, the test points are selected not from an executable program but from formal specification. Because a formal specification defines a system at an abstract level, the cost of domain modeling and test point selection will likely be decreased. [13]

The beginning software testers use the test methods of white-box and black-box testing. To find out the domain errors instead of using similar type of testing the basic methods are used which does not allow the testers to complete the testing in an effective manner. This type of testing methods cannot be guaranteed. To generate test cases that execute specified paths in a program there are many search algorithms as it was said earlier one thing is the genetic algorithm. Other is the simulated annealing algorithm which of all these are used mainly in path testing. [14]

Important technique for the reuse of test tools is the Match technique. This algorithm begins to match from the node of the leaf and look for the test case of using the node of the root progressively, match and generate the test case. In case if any merge or reformation of test case is needed that also can be done in this technique. There are following problems always meet in the test: faced source code that not tested, sequence which test code carrying out are different from the original execution, input the data that not tested, user's operating environment changed. The problem emerges in the test is not go on totally, but many kinds of factors test influences test task, such as time, the human resource and ability. The development of the test case accounts for about 40% testing cycle. The prerequisite of test case reuse: Firstly, there exist test case that could be reused; secondly, the test case reuse must be available; thirdly, testers must know how to reuse the test case. Therefore, whether describe and manage correctly of test case is the key technology in the test case reuse. Generally, a test case always corresponds to test object, specific test goal and relevant test context. The main problem is that how to organize test case and make each test case execute independently effectively. Four principles for automation test of the reuse script of automate test are presented: firstly, to design separate test case independently; secondly, to design the test case including itself; thirdly, to design the test case based on starting point; lastly, to design the test case with non-interval and non-overlap. Among them, the first two principles require have the most cohesive and minimum coupling nature with the test case according to idea of module design in the program. The third principle require no linear dependent relation with initial status merely relates to basic state tested, and for the last principle non-interval require the test case should consist function and system character. Non-overlap means dissipate the redundancy in the test case. The paper focus on the manual test method based on reused test case and black box test. The ability of search test case needs relatively high because the following reasons. The one reason is that black box care behavior of software only to test, and does not care about the logic structure software. The other is generally a text way that descript of the test case is not higher visual degree to the user. The presented algorithm begins to match from the node of the leaf, and look for the test case of using the node of the root progressively, match and generate the test case.

The technique can generate the test case automatically based on the reused test case library and fulfil the development of test case for black-box test. The reused test case can speed up the development of test case. A large amount of suitable test case can be reused for find out and get better coverage rate. [15].

The approach utilizes the advantage of Regression Testing where fewer test cases would lessen time consumption of the testing as a whole. The technique also offers a means to perform test case generation automatically. As for the test cases reduction, the technique uses simple algebraic conditions to assign fixed values to variables (maximum, minimum and constant variables). By doing this, the variables values would be limited within a definite range, resulting in fewer numbers of possible test cases to process. The technique cover all can also be used in program loops and arrays.

- To reduce number of all test cases. Generally, the larger the input domain, the more exhaustive the testing would be. To avoid this problem, a minimum set of test cases needs to be created using an algorithm to select a subset that represents the entire input domain.
- To find the technique for automatic generation of test cases. To reduce the high cost of manual software testing while increasing reliability of the testing processes. With the automatic process, the cost of software development could be significantly reduced.
- To keep a minimum number of test runs. The best technique must be able to generate test cases from only one example test run.

There are four steps to generate test cases:

- Finding all possible constraints from start to finish nodes. A constraint is a pair of algebraic expressions which dictate conditions of variables between start and finish nodes ($>$, $<$, $=$, $\geq$, $\leq$, $\neq$).
- Identifying the variables with maximum and minimum values in the path, if any. Using conditions dictated by the constraints, two variables, one with maximum value and the other with minimum value, can be identified. To reduce the test cases, the maximum variable would be set at the highest value within its range, while assigning the minimum variable at the lowest possible value of its range.
- Finding constant values in the path, if any. When constant values can be found for any variable in the path, the values would then be assigned to the given variables at each node.
- Using all of the above-mentioned values to create a table to present all possible test cases. The proposed technique has achieved greater reduction percentage of the test cases while keeping test cases generation to a single run.

Furthermore, for compilation, it has been found that the technique is the least time consuming among the three. Based on these metrics, the proposed *Coverall algorithm* can be considered a superior technique from all others available in current literatures. Limitation of the *Coverall algorithm* lies in its requirement for identification of fix values for all variables, either as maximum, minimum or constant values. [16]

A test team will use approaches such as these alone or in combination:

- Selection by vulnerability.
- Selection by state changes.
- Selection for path coverage.
- Selection for state coverage.
- Selection by Monte Carlo testing.
- Selection by envelope.

The Dawn mission consists of robotic spacecraft performing a double rendezvous with asteroids Vesta and Ceres over an eleven-year period. Aside from the need for quick solar array deployment and sun-pointing after separation from the launch vehicle, the Dawn spacecraft has no time constrained mission critical events by virtue of it's Ion Propulsion Engines. The encounter with the asteroids will be leisurely compared to the excitement of a Mars entry, descent, and landing activity, for example. The lack of such time critical activities allows Dawn to pursue a basic safe and wait strategy for most fault scenarios, requiring autonomous hardware swapping to cover only the core services needed to maintain safe mode. The method of test case generation for Dawn was part targeting known tricky areas from past experience, and part random failure of key components, especially those critical to S/C health and safety. The goal is to have complete coverage, so as to exercise the entire fault protection subsystem. Tests were generated and prioritized by criticality - since launch was the first key event, it was focused on first, and while its cases ran, other mission phases were added in parallel with the goal in keeping the two dedicated test platforms running 24/7 with the latest FSW version. [17]

The automatic test generation tools we have chosen try to deal with the creation of regression and defect revealing tests. The main difference between those two issues is within the approach used to solve the "oracle problem". In the first case, the tools create tests that characterize the actual behavior of the code. They record and test not what the code is supposed to do, but what it actually does. They consider the tested software as the oracle (i.e. whichever output it produces is correct). We selected two tools (JUnit Factory and Randoop) characterized by good usability and a very reasonable learning curve. On the other hand, the tools which generate defect revealing tests could solve the oracle problem either by asking the user to define what he expects from a class or by relying on language error handling. We wanted to find the easiest and most effective solution available, so we chose two tools: Randoop and JCrasher. While the latter bases its decisions on Java raised exceptions, Randoop only requires the developer to describe invariants he needs on the classes through a Java interface implementation. Hence, we prevented the engineer from wasting time learning new formalisms to describe the tested system. We generated regression tests on the source code present at the end of the manual test implementation. On the contrary we performed automatic defect revealing test generation on the source code as it was before the manual implementation. This permitted a better comparison between manual and automatic test generation approaches. Randoop can generate both regression JUnit tests and error-revealing test cases. The input to Randoop is a set of Java classes to test, a time limit and an optional set of contract checkers. The resulting output is a JUnit test suite. [18]

Sequence-Based Specification (SBS) is a systematic approach to specify a precise mathematical black-box description of systems. SBS consists of several techniques, like *sequence enumeration*, *canonical sequence analysis,* and *sequence abstraction*. After identifying the system's boundary, (i.e) its interfaces with associated stimuli and responses, the sequence enumeration enumerates all relevant stimuli sequences together with their responses and equivalence to prior sequences. SBS can be used to check the completeness and consistency of requirements and to construct a traceably correct black-box specification. Test cases can be generated according to the assigned probabilities. Since the generated test cases can be seen as a random and representative sample of the expected use of the system, the test case results can be statistically analyzed and can be used for reliability estimations of operational use. Transition probabilities are used for test case selection and can also be used to model safety-critical usage. In this case, the generated test cases can be used to analyze the safety properties of a system.

The Taguchi Method is a quality engineering method that considers noise factors (environmental variation during the product's usage, manufacturing variations, and component deterioration) and the cost of failure in the field. The Taguchi Method was extended for test case generation by selecting a combination of parameters using an orthogonal array. The generated test cases make it possible to detect all double-mode faults. If a consistent problem exists when specific levels of two parameters occur together, this is called a double-mode fault. A double-mode fault is not a highly probabilistic fault; however, we believe double-mode faults are related to the unexpected failure of a system reports that a testing strategy using the Taguchi Method is more effective than previous testing strategies. [19]

The first successful model checking approach was explicit model checking, which performs an explicit search in a model's state space, considering one state at a time. The search might be based on a breadth first search (BFS), depth-first search (DFS) or possibly also heuristic search algorithm. There are several different approaches based on different temporal logics. The main concern when using model checkers for this task is the performance, which can be problematic due to the state space explosion. In general it is difficult to predict how a particular model checking technique will perform for any given specification. Consequently it is impossible to give detailed rules about which model checker to use for a given specification. [20]

SpecTRM-RL notation is formal, yet the syntax is similar to English and easy to learn to understand. A SpecTRM-RL model describes system inputs, outputs, state values, and internal modes. A state value represents information inferred by the system regarding the current operating environment. Internal modes represent different collections of behavior. For example, a system in the mode "Waiting for Liftoff" would respond differently than it would when in the mode "Initiating Landing". Several algorithms were developed for generating sets of test cases. These algorithms vary in the number of test cases they generate, trading efficiency for robustness. A fully comprehensive test suite would generally include so many tests that it would be impractical to use. It is possible to generate much more manageable suites of test cases that are still effective at detecting the majority of software defects. Each of these algorithms identifies a set of scenarios to be tested, where a scenario is defined to be a set of conditions that should lead to a specific system state. Once these scenarios are identified, it is necessary to also generate sequences of inputs that would satisfy the scenario. Safe ware has developed algorithms to automatically generate test cases directly from SpecTRM-RL requirements models. Seven algorithms were considered for test case selection. Four of these algorithms were designed to test whether the system makes internal transitions appropriately in response to inputs. The other three algorithms were each designed to detect a specific type of software defect. [21]

Author has reported on investigations regarding the structure of test suites. In particular it is of interest how the length of individual test cases influences the characteristics of a test suite. To analyze this we have performed a set of experiments, where the average length of test cases is continually increased and the effects on test suite size, length, coverage, redundancy, minimization, and monitoring were observed. When deciding whether to prefer long or short test cases there are many different special cases depending on the testing environment that need to be considered. For example, longer test cases are counterproductive if minimization is applied as a post-processing step. However, in most scenarios it seems feasible to give preference to fewer longer test cases instead of many short test cases: In fact an increase in test case length can reduce the overall size and length of the resulting test suites while actually increasing the fault detecting capability at the same time. [22]

The first approach to the field of Object-Oriented Evolutionary Testing, based on the concept of Genetic Algorithms, was presented by Tonella in 2004. In this work, the eToc tool for the Evolutionary Testing of Object-Oriented software was described. The approach presented involved generating input sequences for the white-box testing of classes by means of Genetic Algorithms, with possible solutions being represented as chromosomes. A source-code representation was used, and an original evolutionary algorithm, with special evolutionary operators for recombination and mutation on a statement level – i.e., mutation operators inserted or removed methods from a test program was defined. A population of individuals, representing the test cases, was evolved in order to increase a measure of fitness, accounting for the ability of the test cases to satisfy a coverage criterion of choice. New test cases were generated as long as there were targets to be covered or a maximum execution time was reached. However, the encapsulation problem was not addressed, and this proposal only dealt with a simple state problem. Our evolutionary approach for automatic test case generation is described. The concepts presented were implemented into the eCrash automated test case generation tool for Object- Oriented Java software. Additionally, an Input Domain Reduction methodology, based on the concept of Parameter Purity Analysis, for eliminating irrelevant variables from Object-Oriented test case generation search problems was proposed. With our approach, test cases are evolved using the Strongly-Typed Genetic Programming paradigm; Purity Analysis is particularly useful in this context, as it provides a means to automatically identify and remove Function Set entries that do not contribute to the definition of interesting test scenarios. Nevertheless, the concepts presented are generic and may be employed to enhance other search-based test case generation methodologies in a systematic and straight-forward manner. The observations made indicate that the Input Domain Reduction strategy presented has a highly positive effect on the efficiency of the test case generation algorithm; less computational time is spent to achieve results. [23]

The likely invariants extracted from executing a test suite, such as the loop invariants, pre-conditions or post-conditions, could uncover the program properties to some extent. While the invariants extracted from executing a new test case *NTC* are different from the ones extracted without executing *NTC*, the test case *NTC* could reveal some new properties or cover some new paths. So the test case *NTC* is effective and is added to test suite. While the invariants remain unchanged, it indicates that the test case could not reveal any new property, so the test case should be discarded. We integrate the generator and the invariant extraction technique to generate and select test cases according to the changes of the invariants. Finally we can automatically generate test suite with high quality and moderate size. We analyze the effect of different values of *CN* through the experiment, and verify that, compared with traditional random test case generation technique, our technique could generate smaller test suite with the same invariants. This reaches the goal of reducing the workload and cost of software testing. [24]

Orthogonal test method is a kind of designing method to research many factors and levels. It conducts tests by from selecting a representative sample of test points, which have evenly dispersed, neat comparable characteristics, according to orthogonality from comprehensive tests. The design of orthogonal experimental is based on the orthogonal table, efficient, rapid and economic method of the experimental design. Orthogonal test method is a kind of scientific method which would select a suitable number of representative test cases from many test data then to arrange test reasonably. When designing test cases by using orthogonal test method, first of all, we should find the objects of impacting its function according to the instructions of being tested software, put the objects as factors, and put all the factors as the level of different value. The basic idea of greedy algorithm is from an initial solution of the problem to a given goal successively, to seek a better solution as fast as possible. When one step of the algorithm cannot continue to move forward, the algorithm stops. Greedy algorithm a kind of the classification treatment method improved, the characteristics of which are to carry on in according to one optimized measure step by step, and every step must be guaranteed to get access to the local optimal solution. Greedy algorithm is not a particular algorithm, but rather a kind of abstract one. Its specific performance does not search for the solution space mechanically, but selects the better local. It does not search solution according to this greedy algorithm strategy until completing all of the solutions, in order to find a viable solution to meet the greedy strategy efficiently. The steps of optimization of test cases using Greedy algorithm are as follows:

- Setting up the set of test demand
- Test Set Optimization.

This method reached the purpose of optimization of test cases. [25]

Decision table-based testing is a black-box or functional testing technique. In black-box approach test data are derived from the specified functional requirements without regard to the final program structure. It is based on the view that any program can be considered to be a function that maps values from its input domain to values in its output range. Decision table-based testing is closely related to, and in some sense has evolved from other functional techniques like Equivalence Class Testing and Boundary Value Testing. [26]

The principle of ART (Adaptive Random Testing) is to evenly spread test cases. This principle can be implemented in different ways and, therefore, several ART algorithms have been developed. The first ART algorithm proposed is known as the *Fixed Size Candidate Set ART* (FSCS-ART). In this algorithm, an initial test case is randomly chosen and run. Then, to choose a new test case, a fixed number of candidates are randomly generated. A concern with ART is its time complexity in test case selection. Compared with test case generation, however, it is often more expensive or time consuming to run a test or to verify a test result. In these situations, it is highly desirable to have a strategy that can reduce the number of required test case executions, and ART helps to achieve this goal. [27].

## III. CONCLUSIONS

This paper has presented the various algorithms, tools and methods which have been developed by the researchers so far in the area of detecting the domain errors in Object-oriented programs. This will be very useful for the upcoming developers and testers in software testing. Through the years a number of different methods have been proposed for generating test cases it can also be derived from system requirements. One of the advantages of producing test cases is that they can be created earlier in the development life cycle and be ready for use before the programs are constructed. Additionally, when the test cases are generated early, Software Engineers can often find contradiction and uncertainty in the requirements specification and design documents. This will definitely bring down the cost of building the software systems as errors are eliminated early during the life cycle. Further, we would choose any one of the techniques and develop them to overcome the limitations in a better way.

## REFERENCES

[1] M.L.Shooman and M.I.Bolsky member IEEE, "Types, Distribution and Test and correction times for programming errors" on IEEE transaction on reliability, Vol R-25, No.2, June 1976.

[2] Jie Chen, Member, IEEE, "Frequency-Domain Tests for Validation of Linear Fractional Uncertain Models" IEEE TRANSACTIONS ON AUTOMATIC CONTROL, VOL. 42, NO. 6, JUNE 1997.

[3] Robert McNaughton, "Contributions of Ronald V. Book to of string-rewriting systems" Theoretical Computer Science the theory Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180-3590. USA Theoretical Computer Science 207 (1998) 13-23.

[4] Bingchiang Jeng, "Toward an integration of dataflow and domain testing" Department of Information Management, National Sun Yat-Sen University, Kaohsiung, Taiwan 80424, ROC 1999 Elsevier Science Inc.

[5] MIA Huaikou and LIU Ling, "A Test Class Framework for Generating Test Cases from Z Specifications" 2000 IEEE.

[6] Karl R.P.H. Leung4 Wai Wongv, "Deriving Test Cases Using Class Vectors" 2000 IEEE.

[7] Yu-Wen Tung and Wafa S. Aldiwan, "Automating Test Case Generation for the New Generation Mission Software System" 2000 IEEE.

[8] Joakim Aidemark, Peter Folkesson, and Johan Karlsson, "Path-Based Error Coverage Prediction" 2001 IEEE.

[9] Sudheendra Hongal and Monica S.Lam, "Tracking down software bugs using automatic anamoly detection" in ACM transactions ICSE may 2002.

[10] T.Y. Chen, Pak-Lok Poon, Member, IEEE, and T.H. Tse, Senior Member, IEEE, "A Choice Relation Framework for Supporting Category-Partition Test Case Generation" 2003 IEEE.

[11] Patrick J. Schroeder, Eok Kim, Jerry Arshem, Pankaj Bolaki, "Combining Behavior and Data Modeling in Automated Test Case Generation" Proceedings of the Third International Conference On Quality Software (QSIC'03) 2003 IEEE.

[12] Ruilian ZhaoMichael R. LyuYinghua Min, "Domain Testing Based on Character String Predicate" Proceedings of the 12th Asian Test Symposium (ATS'03) 2003 IEEE.

[13] Yuting Chen and Shaoying LIU, Proceedings of 11th Asia pacific software engineering conference (APSEC 04) IEEE.

[14] Nasha Miran, "Data Generation for Path Testing" Software Quality Journal Kluwer academic publisher's pg 121-136, 2004.

[15] Zhenyu Liu and Ning Gu, "An Automate Test Case Generation Approach Using Match Technique" The Fifth International Conference on Computer and Information Technology IEEE.

[16] Preeyavis and Jiranpun, "Coverall Algorithm for Test Case Reduction" IEEEAC, IEEE transactions 2005.

[17] Kevin J. Barltrop, Kenneth H. Friberg, Gregory A. Horvath "Automated Generation and Assessment of Autonomous Systems Test Cases" IEEE 2007.

[18] Alberto Bacchelli, Paolo Ciancarini and Davide Rossi, "On the effectiveness of manual and automatic unit test generation" The Third International Conference on Software Engineering Advances 2008 IEEE.

[19] Haruka Nakao and Robert Eschbach, "Strategic usage of test case generation by combining two test case generation approaches" The Second International Conference on Secure System Integration and Reliability Improvement IEEE 2008.

[20] Gordon Fraser and Angelo Gargantini, "An Evaluation of Model Checkers for Specification Based Test Case Generation" 2009 International Conference on Software Testing Verification and Validation IEEE.

[21] Kenneth Kelley, "Automated Test Case Generation from Correct and Complete System Requirements Models" IEEE 2009.

[22] Gordon Fraser and Angelo Gargantini, "Experiments on the Test Case Length in Specification Based Test Case Generation" AST'09 2009 IEEE.

[23] José Carlos Bregieiro Ribeiro , Mário Alberto Zenha-Rela, Francisco Fernández de Vega, "Test Case Evaluation and Input Domain Reduction strategies for the Evolutionary Testing of Object-Oriented software" Information and Software Technology 51 (2009) 1534–1548 Elsevier.

[24] Fanping Zeng, Qing Cao, Liangliang Mao and Zhide Chen, "Test Case Generation based on Invariant Extraction" 2009 IEEE.

[25] Kenneth Kelley, "The research of test case generation and its optimization methods based on orthogonal test method and greedy algorithm" International conference on Intelligent Human machine system and cybernetics 2009.

[26] Mamta Sharma and Subhash Chandra B, "Automatic Generation of Test Suites from Decision Table - Theory and Implementation" 2010 Fifth International Conference on Software Engineering Advances IEEE.

[27] Zhi Quan Zhou, "Using Coverage Information to Guide Test Case Selection in Adaptive Random Testing" 2010 34th Annual IEEE Computer Software and Applications Conference Workshops.