Chapter 2

Automated Software Test Data Generation: Direction of Research

2.1 Introduction

A challenge for IT industry is to develop software system that meets business needs. The fact is we are to deliver software that are free of bugs. The bugs in software can cause major loss in IT organization if they are not removed before delivery. Software testing is an important parameter developing software that are free from bugs and defects. Software testing is performed to support quality assurance [21]. A good quality software can be made by using an efficient test method. Statistics say that 50% of the total cost of software development is devoted to software testing even it is more in case of critical software[22]. Depending on time, scale and performing methods we can classify testing as unit testing, integration testing, system testing, alpha testing, beta testing, acceptance testing, regression testing, mutation testing, performance testing, stress testing etc. There are testing like statistical testing which is used for measuring reliability of software rather than finding errors. Test data can be generated either based on specification [27, 28, 23] or code. In the literature of automated test data generation, searched based data generation survey is available[24]. Although code based survey for test data generation has been discussed by many authors [22, 21] yet there is a field to study in context of program analyzers, test data generation models etc. In this chapter, we basically concentrate on survey of code based [22] test data generation. Test data can be designed either manually or automatically. Software engineering research puts large emphasis on automating the software development process that produces large more complex quantities of code with less effort. For testing these software, we need to find advance innovative support procedures to automate the testing process[5]. In spite of continuous effort till today automated testing has limited impact in industry, where the test generation activity remains largely manual. What we need is 100% automated testing to reduce overall cost of software development with high quality. A number of test data generation techniques such as random test data generator, path oriented test data generator, goal oriented test data generator, and intelligent test data generator have been automated. Nowadays testing on networking environment, testing to improve the scalability of software testing is emphasized [5].

Test data generation research are going on since 1970's. But unfortunately till today there is hardly any fully automated test data generation tool found in industry. Initially people did research on test data generation using symbolic execution in 70's upto mid 80's [29, 30, 1]. At that time the language taken for test data generation was FORTRAN Algorithm [15]. In 1987 Parther [25] had contributed a new idea for test data generation called path prefix method. In 1990, B. Korel had made a revolutionary change by generating test data dynamically based on actual value using pattern and explanatory search [3]. In 1996 Korel [31] developed assertion oriented and chaining approach [32], Goal oriented test data generation. In 2000 test data generation on dynamic data structure is emphasized[33, 34, 35, 36, 37]. Mahmood in

[10] has given a good review of test data generation techniques from 1997 to 2006. But the paper ignores technical details of the methods found in his reviewed paper. During 2004 to 2006 clever implementation of random testing is done to get the benefits of avoidance of infeasible paths and to ignore path selector module[38, 4, 39]. During this time test data generation using hybrid method that takes the advantages of both static and dynamic method were done [11, 12]. The William work, 'path crawler' has the advantages because it ignores infeasible paths. But problem is of exponential increase of number of paths. In 2005 Chen et al [40] had shown how to implement automated test data generation for teaching students. This type of work is very useful for beginners to know how to implement research paper. In 2007-08 there are many other papers who worked to detect infeasible paths for saving computational time [7, 6]. In 2010 Tahbildar etal [41] has given a heuristic to determine the number of iteration required for longest path coverage. In 2000-2010 Object oriented test data generation techniques are also taken as key area of research. Most of the industries are using object oriented techniques for software development due to high productivity. UML has got a great importance for software testing of object oriented programs. Different UML diagrams are used for different types of testing[42, 43, 44]. Mutation testing is used to improve reliability of object oriented software[45]. A scalable test data generation based on multi agent is proposed by Siwen in [46]. In 2008 Xio [13] has proposed a method for guaranteed test data generation even if some path predicate are unsolvable. But the method could not give a good coverage. In 2008 Gautam Saha [9] made a map which gives us clear understanding of software testing types, concepts and their relationship which may help researchers to get a complete

picture of the domain. In 2010 [47] proposed a heuristic specially useful programming construct having loops of different dimensions and array of variable length. Moreover, in 2010 [48] test data is generated by avoiding unsolvable constraints.

The chapter is organized as follows: in section 2.2 basic concept related to software testing, different types of customer requirements, Architecture of test data generation, Program Analyzers, and path selector are explained. Section 2.3 discusses different test data generation techniques with their relative merits and demerits. Section 2.4 states different implementation techniques for automated test data generation. Survey related works are explained in section 2.5. Section 2.6 discusses about the future challenges and problems that are required to solve for generating test data efficiently. Section 2.7 states about future trends of test data generation by listing some problems to be solved in automated test data generation. Finally we conclude with concluding remarks in section 2.8.

2.2 Basic concepts

2.2.1 Concept related to software testing

Software testing is the process of ensuring right software product to achieve full customer satisfaction. The following terms are mostly used for automated test data generation research [49].

Test data: Test Data are data which have been specifically identified for use in testing computer program.

Test case: A test case is a set of conditions or variables under which a tester will determine whether an application or software system is working correctly or not.

Test oracle: The mechanism for determining whether a software program or system has passed or failed such a test is known as a test oracle.

Test suite: A set of test cases is called test suite.

Test plan: is a document which contains all the information about the testing of all stages.

Test Automation: Developing software for testing a software product.

Coverage: Coverage of program or faults. The aim of coverage based testing method is to 'cover' the program with test cases that satisfy some fixed coverage criteria. The aim of fault coverage criteria is to cover maximum fault in a program.

Path: sequence of nodes and edges. If we start from entry node and end at exit node then we call complete path.

Branch predicate: is a condition in a node that may lead to either true path or false path.

Path predicate: A path predicate is defined as the collection of branch predicate which require to be true in order to traverse a path.

Feasible path: The path where there is valid input that execute in the path.

Infeasible path: The path where there is no valid input that execute in the path.

Constraint: A constraint is an expression that specifies the semantics of an element, and it must always be true.

Constraint generator: is a program that can automatically generate the set of conditions or constraints in a path.

Constraint solver: is a program that provides value to the input variables of a path predicate such that it satisfies all the constraints of the path predicate at a time.

Constraint programming: Constraint programming is a programming paradigm where relations between variables are stated in the form of constraints.

Softwares are tested or validated based on functional requirements, non functional requirements, and business requirements [50].

Functional requirements

Functional requirements are associated with specific functions, tasks the software must support, what are the works software supposed to do. It deals with product features and product functionality.

Nonfunctional requirements

Non-functional requirements are constraints on various attributes of functions or tasks. It deals with quality of product. Testing for maximum ability, speed, efficiency, reliability, safety, and scalability etc. For example how many users can simultaneously try to vote in a online voting software where millions of voters may caste vote on the same day. Testing these type software manually is not possible. Non functional testings are key area for automation where manual testing is impossible. Both functional and non functional requirements are included in software requirement specification document.

Business requirements

Business requirements are the basic requirements of customer that are to be fulfilled for smooth running of their day to day work. Neither functional nor non functional they know, but they are interested only on their business. They always mean business running smoothly which implies that software requirements are fulfilled. It is not included in software requirement specification document.

Software testing plays an important role in developing software that are free from bugs and defects. It consists of three major steps: (i) Designing test cases i.e. generating data(test data) for the input variables, (ii) Executing the software with those test cases and (iii) Examining results whether it is as per requirements written in SRS (software requirement specification) document. It is observed that all new inventions are passed through a series of different tests based on well-defined criterion in order to verify correctness against specification, determination of performance and quality of the product.

We classify testing work with words *How*, *When*, and *Who*.

How- By the word how we mean how test data is generated. Testing can be done either based on code, that may be source code, executables, binaries or based on experience, previous knowledge about input and output of a particular code. In code based testing basically we try to cover maximum code to minimize error by fulfilling criteria like statement coverage, branch coverage, condition coverage, and path coverage. We call these type of testing as white box testing. White box testing is done by technical persons/developers who has knowledge of programming. Testing based on experience, considering the program as black box is called black box testing. There are different methods of black box testing like equivalence partitioning, boundary value analysis etc., for example the program to check palindrome. We may put appropriate data and check the reliability of white box testing which is based on probability. For example, the test data covers 95% of code, but still the program may fail if the test data is belongs to 5% of code.

When- By the word when we mean different time of testing. Whether testing is done in only testing a phase or all phases of software development. Accordingly, testing is classified as verification and validation. Verification mean testing during all phases basically static testing without executing with test input. Validation testing is done in testing phase i.e. testing with executing test inputs.

Who- By the word who we mean who does the testing. There are three types of testers. Developers/engineers, friendly customers and actual customers. These testings are called alpha, beta, and acceptance testing respectively. A schematic view of software testing is given in figure 2.1.

Designing Test data or generating test input can be done either manually or automatically. Test data generation is a complex activity because it involves so many sub-steps and each sub-step has several related issues. Different authors had proposed different models representing architecture of test data generation. Based on test data generation implementation technique, we have found following architectures are commonly used in software test data generation literature.

Architecture I based on symbolic value execution

Schematic representation of the architecture is shown in figure 2.2. The model is based on the concept of symbolic execution. It consists of three parts: path selector, constraint generator and constraint solver. The path selector generates a set of paths from the input program satisfying some criteria. One of the criteria is statement coverage i.e. we need to select the set of paths in such a way that each and every statement is covered at least once. This is poor criterion because all branches in the program may not be exercised and errors may remain undetected. A stronger criterion is the branch coverage i.e. selecting the set of path covering all branches in the program so that all statements as well as branches are executed at least once. Another interesting criterion is the testing of boundary value. Here we need to select the path in such a way that it will ensures all branches to be covered by at least one path. For each loop, there may be one path covering the loop at least once and another path that does not cover the loop at all. The constraint generator creates the path constraints either from the source program directly or from the test path generated by the path selector. Concept of symbolic execution is applied to find the constraints. Symbolic execution means executing the program using some symbolic variables instead of actual value of input variable. Symbolic execution can be performed in two ways: forward traversal (forward substitution) and backward traversal (backward substitution). In forward substitution, symbolic execution is performed on every executable statement and intermediate symbolic values are stored for subsequent use. Number of paths may grow very rapidly in this approach. In backward substitution, path is traversed in reverse order. It generates path constraint in a bottom-up fashion by first finding constraints for the called routine. This approach is suitable for testing large programs. Here advantage is that no storage is required to store the symbolic values of variables. On the other hand forward substitution can detect infeasible paths early with contradicting input constraints. In handling array also it has advantage over backward substitution. The output of constraint generator is the path constraint. Each path constrain is a set of equalities and inequalities on the input variables and the set of values that satisfy these constraints which are the required set of test data for the respective path. The set of test data is found by constraint solver. The constraints may be linear or non-linear. Depending upon the type of constraint it applies different techniques to solve these constraints. For linear constraints, linear programming techniques are used. For nonlinear constraints, non-linear programming techniques need to be applied. Using systematic trial and error method also test data can be generated. In [48] Tahbildar and Kalita have proposed a model for test data generation where they consider only the solvable constraints. It avoids unsolvable constraints.

This architecture is best suited for programs with less number of constraints and if they are less correlated.

Architecture II based on concrete/actual value execution

Schematic representation of the architecture is shown in figure 2.3 [14]. It consists of three parts: program analyzer, path selector and test data generator.

2.2.2 Program Analyzer

It analyzes the source code automatically. Program analyzer takes a piece of software as input and produces necessary data to be used by path selector and/or test data generator. For analyzing program, control flow graph, data flow graph, data dependence graph, program dependence graph and extended finite state machines are most commonly used.

Control Flow Graph (CFG)

CFG describes the sequence in which the statements/instructions of a program are executed. It is a representation of flow of control through the program. CFG is directed graph in which each node is a program statement/basic block and each edge represents the flow of control between statement/basic blocks. A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibly of branching except at the end. In [22] CFG is defined as a directed graph G=(N,E,s,e) consisting of a set of nodes N and a set of edges $E = \{(n, m)|n, m \epsilon N\}$ connecting the nodes. All edges are labeled with a condition or a branch predicate. If a node has more than one outgoing edge the node represents a condition and the edge represents branch. CFG has two special nodes: one entry(s) and one exit(e) node. Control flow information is used in Path Oriented Testing, Goal Oriented testing, and Random testing. Figure 2.4 shows the control flow graph for GCD computation program listed in Appendix A. This architecture is best suited for programs having pointer data types, constraints with different variables.

Architecture III based on both symbolic and actual value execution

Schematic representation of the architecture is shown in figure 2.5. It consists of program analyzer, code instrumentor, filter, and comparator. The code instrumentor statically inserts lines to the source code to display the paths and the filter provides us the only unique feasible paths. The comparator compares the number of paths in current iteration and previous iteration to terminate with minimum number of iterations either for longest path criteria or all path criteria. Details of this model can be seen in [47]. This architecture is best suited for programs having loops with variable number of iterations and array with different dimensions.

Architecture IV based on object oriented approach

The schematic representation of architecture is shown in figure 2.6. In this model [17] components are system model written in UML diagrams, and test directives written in UML. The compilers produce state machine written in a intermediate format language. The output of the state machine is an abstract test suite containing sequence of simulations and observations. This model is useful for non functional model based testing like scalability, stress testing etc. This architecture is best suited for programs with object oriented approach and for performing system testing.

Data Dependence Graph (DDG)

According to [51], DDG is a graph that represents data dependencies between statements. Nodes in the graph represent memory references and edges represent data dependencies between nodes. Before building DDG we need to construct CFG. DDG can be generated from CFG in two ways. One is to generate DDG directly from CFG using the information of data dependence. Data dependency can be defined as follows. Let the set DEF(i) and REF(i) denote the sets of variable defined and referenced at node i of the CFG. Node j is data dependent on node i if there exist a variable x such that: i) x DEF(i) ii) x REF(j), and iii) there exist a path from node i to j without intervening definition of x.

The other is first to build Program Dependence Graph (PDG) and then convert PDG into DDG by deleting the control dependence. Both the data and control dependencies for each operation in a program are explicitly shown in the PDG. Data dependence exists between statements S1 and S2 if S1 defines a variable and S2 has a reference to the variable and there is a path in the program from S1 to S2 on which the variable is not defined again, which means the definition of the variable in S1 reaches the use in S2. If statement S2 is dependent on statement S1, then (S1, S2) is a definition-use pair. We can get DDG by deleting the control dependence relations between the statements. Figure 2.7 shows the data dependence graph for GCD computation program listed in Appendix A. Data-dependence graph defines a partial order between the operations performed by a program [52]. When a reordering of the program's operations or instructions does not inverse the DDG arrows, then the semantic of the program is preserved. The DDG is a static information: it relates textual operations from the program. When the operation is enclosed in a control structure, such as a loop, it represents all its run-time executions.

Program Dependence Graph (PDG)

The PDG represents a program as a graph in which the nodes are statements, predicate expressions, and the edges incident to a node represent both data dependence and control dependence. More than CFG, the PDG can present the dependence of any statements, including data and control dependence, not only the control flow. To build PDG first CFG is constructed. The next task is finding data dependence and control dependence in CFG. Figure 2.8 shows the program dependance graph for GCD computation program listed in Appendix A.

Extended Finite State Machine (EFSM)

In program dependence graph or data dependance graph some information are not available but in EFSM almost all information in a program is represented. According to [18] EFSM can be defined as (S,V,T) where S is the set of states, V is the set of variables, T is a set of transitions. S contains one initial state and one or more final states which represents end of program execution. The EFSM moves from current state to next state by updating the values of variables performing some action. In action it executes input statements or assignment statements that gives values to variables. In [18] Zhang uses EFSM for selecting the set of paths. Figure 2.9 shows the EFSM for GCD computation program listed in Appendix A.

The tools discussed above are seen to be used extensively by different researches for automated test data generation. Different methods using either actual value or symbolic value approach are found in the literature that takes the advantage of using CFG in the process of automated test data generation. In [39] Gotlieb and Petit used CFG for selecting path and then apply backward symbolic execution. Xiao Ma et al. [13] have used CFG in their constraint prioritization method using which test data can be generated even in a situation when constraints are not solvable. In path prefix strategy [25] also CFG has been used to select new path from using previous path as a guide to the selection of subsequent paths. In [53] Zhang et al. extract path conditions from CFG and then use a tool to decide satisfiability of the constraints. In binary search based test data generation technique also CFG has been used for selecting paths. Data Dependence Graph and Program Dependence Graph are found to be used mainly in program optimization. For instance Xinyn Wang et al [51] uses DDG for the purpose of automatic identification of domain variables from source code and Laurent Hascoet applied DDG for computing gradients in program optimization. PDG has been used by [52] as a program representation tool for program optimization and [54] uses PDG for generating sequential code from concurrent specification in Esterel V5 compiler. EFSM has been used by [18] for generating test path using symbolic execution and [55] describes a method of generating EFSM whose output may be used by any EFSM analyzer. The suitable mapping between program analyzers and test data generation methods are shown in figure 2.10.

2.2.3 Path Selector

The path selector identifies a set of paths to satisfy selected testing criterion. A good path selector ensures high coverage of code. Basically we are interested in basis set of paths as the number of program paths in a program may be unexpectedly high and sometimes impossible to test all the paths of a program[14]. A basis set of path should be linearly independent covering all edges of CFG and the basis set path can derive all possible path by linear operation on the basis set paths. This phase is ignored in many recent test data generation techniques.

2.3 Test Data Generator (TDG)

The Test data generator utilizes the information generated by the program analyzer and /or path selector. Once path information is obtained, aim of TDG is to find input values that will traverse a specific path. This is achieved by finding the path predicate and then solving it in terms of input variables. To solve the path predicate representing a system of inequalities, various search method such as alternating variable, simulated annealing and different heuristics based on equation-rewriting can be applied.

2.3.1 Random Test Data generator

Random test data generation takes test inputs at random until a useful input is found. According to [39], random testing is the process of selecting test at random according to an uniform probability distribution over programs input domain. This approach is quick and simple. But disadvantage is poor code coverage. Now a days several researchers are proposing clever implementations of random testing that can improve code coverage like systematic test generation. The random testing can be implemented either based on symbolic value [1, 29, 18, 53, 18] or actual value [56, 4, 38]. Symbolic execution is useful technique for verification and testing. Using symbolic execution, we can collect a path predicate which is a sequence of branch predicate appeared in a particular path. The set of constraints found can be solved using constraint solver to get test data. Therefore the efficiency of the constraint solver is the main issue. This method requires no constraint violation checks of branch predicates since all can be solved at once. But the problem in this method is that a variable value in symbol may be a very complex expression and difficult to solve. The situation becomes more problematic if non-linear operator appears. In actual execution instead of variable substitution we run the program with some, possibly, randomly selected input. The objective here is to modify the initial randomly selected input so that the intended path is taken. This method is quite expensive in terms of speed of execution. It requires backtracking. The main challenge of this method is efficiency of search and to select appropriate heuristics so that the number of backtracking is minimized. At present people are emphasizing to develop hybrid approach which combines the gains of both symbolic and actual based methods. In random testing recent research uses hybrid approach for test data generation. In [39] random testing is implemented based on actual value using symbolic reasoning. DART (directed automated random testing) means directed search. Starting with a random input, a DART implemented program calculates during each execution an input vector for the next execution. This vector contains values that are the solution of symbolic constraints gathered from predicates in branch statements during the previous execution. The new input vector attempts to force execution of the program through a new path. By repeating this process directed search attempts to force the program to sweep through all its feasible execution paths. The challenge in DART is to solve the constraints generated by a program. Kousic sen's Concolic testing extended the DART works for programs which have dynamic data structure using pointer operation[4]. CUTE implements a solver for both arithmetic and pointer constraints to incrementally generate test inputs. But main demerits of this testing method is that information about test requirement is not incorporated into the generation process, hence it may fail to satisfy the requirement. Also it has poor coverage and has difficulty in finding semantically small fault.

2.3.2 Goal Oriented Test Data Generator

Korel [32] defined the goal-oriented approach of test-data generation as the process of generating input test data to execute the selected statement irrespective of path taken i.e., the path selection stage is eliminated. It uses data dependencies to guide the search process. Data dependence analysis is used to identify statements that affect execution of the selected statement. It generates input that traverses a given unspecific path. Informally an unspecific path is a path with some segments missing. Since this method uses the find-any-path concept, it is hard to predict the coverage given a set of goals.

Two typical approaches, 'Assertion-based'[31] and 'Chaining approach'[32] are known as goal oriented. In the first case assertions are inserted and then solved while in the second case data dependence analysis is carried out. Generally the goal-oriented approach faces issues of goal selection and selection of adequate test data.

2.3.3 Intelligent Approach

The intelligent test-data generation approach often relies on sophisticated analysis of the code to guide the search for new test data [57, 58]. However this approach can be extended up to the intelligent analysis of program specification as well. With the proposed extended ability this approach will fall in between functional and structural testing. At this time this approach is quite limited in use therefore, its specialization in use or its pros and cons cannot be stated with any certainty.

2.3.4 Path Oriented Test Data Generator

The path oriented test data generator basically involved 3 steps : program analyzer, path selector and test data generator. There are various implementation methods for path oriented testing. In 1976 Clarke [1] implemented path oriented testing using symbolic values on a given path and used linear programming technique to solve the linear path constraints. Biggest problem was how to handle the problem of infeasible paths and complex constraints. In 1987 Prather [25] generated test data using adaptive method called path prefix strategy which avoid path selector step and utilize the best of the previously traversed paths at each selection of a new test input. In 1990, B. Korel[3] generate test data using actual value using heuristic function minimization techniques to modify the input so that a path is covered. Though Korel uses some heuristics for expediting the search process but still this method is costly in terms of computation specially in presence of infeasible paths. The focus of the above test data generation methods was on basic data types such as integer and real. In [34, 2] the test data generation methods was extended for dynamic data structure. Jian and Xiaoxu [53] emphasized over improvement of constraint solver in path oriented testing by including boolean expression. Jian [18] gives a path oriented approach based on combination of symbolic execution and constraint solving. Xiao in [13] proposes a constraint prioritization method using data sampling scores to generate test data even the set of constraints is unsolvable. In [11, 12], PathCrawler generated path test by combining static and dynamic analysis. In path-oriented test data generation the typical approach is generation of a control-flow graph. In this approach, at first a graph is generated and subsequently, by using the graph a particular path is selected. With the help of a technique such as symbolic evaluation (in the static case otherwise it is called function minimization) test data is generated for that path in the end. Many white box testing methods can be viewed as path oriented. A set of paths of the program are selected, such that some criteria are met. This method is more reliable but computation cost is very high. Path oriented testing can be implemented using symbolic value [1] or actual value [59, 3]. In [3] B. Korel uses heuristics in function minimization search for reducing the constraints violation and backtracking. In fact, path testing is very costly for large program containing loops. The number of paths is unbounded and deciding whether a path is feasible or not is an undecidable problem in general case. A subset of paths can provide good code coverage. In real life program a large portion of the paths cannot be executed [18]. Therefore path testing method should be developed that only generate test data for feasible paths. The cost of path testing can be reduced by avoiding path selection step by using path prefix technique [25, 12] or test prioritization technique [60, 56, 61]. Therefore more research is required for selecting only feasible paths, handling loops and to avoid reliance on a set of preselected complete paths to be traversed for test data generation.

2.4 Implementation methods

2.4.1 Static method(using symbolic execution)

These are the testing methods adopted for analysis and checking of system representations such as the requirements documents, design diagrams, and the software source code, either manually or automatically, without actually executing the software. In other words, the static methods do not require the software under test to be executed. They generally use symbolic execution to obtain constraints on input variables for the particular test criterion. Solutions to these constraints represent the test-data. Executing a program symbolically means that instead of using actual values, 'variable substitution' is used.

Static approaches to test case generation [1] selects a path from the control flow graph for test requirement, derive path predicate as a set of constraints [2] on the input symbolic values and then solve the constraints [62] to find a test case which executes the path. Static generation suffers problem to detect infeasible paths in case of loops with a variable number of iterations. In general static technique is vastly weaker than dynamic at gathering the type information needed to generate real test cases[59]. It is useful only for straight forward code. Main difficulty in this technique is to solve the non linear constraints.

2.4.2 Dynamic method(using actual value)

Instead of using variable substitution, these methods execute the software under test with some, possibly randomly selected input. By monitoring the program flow the system can determine whether the intended path was taken or not. In case of a negative answer the system backtracks to the node where the flow took the wrong direction. Using different kinds of search methods the flow can then be altered by manipulating the input in a way so that the intended branch is taken. In [63] Gupta combine symbolic reasoning with dynamic execution. But scalability of these techniques are poor. In [3] B. Korel used heuristic function minimization techniques to modify the input so that a path is covered. But this approaches suffer from many problems, such as the number of execution may be more and it may fail to find test case even if one exist. Finally, in case of infeasible path it will not terminate.

2.4.3 Hybrid Implementation

Combining static and Dynamic test data generation The recent research on test data generation emphasizes on hybrid approach taking advantages of different methods of test data generation. In [12] N. Williams work objective is to have 100% coverage of feasible execution paths. The method is not following the traditional steps of test data generation instead, iteratively cover on the fly the whole input space of the program under test. It takes path prefix partial path predicate and solve using constraint logic programming. The method tries to avoids problems of both complexity of static analysis and number of executions required in heuristics algorithms [3] used in function minimization.

2.5 Related Works

In the literature, survey of test data generation [21, 22, 24, 5, 10, 9, 23] are classified into specification based survey [23] , code based survey [22], and searched based survey [24]. Harrold survey focuses on software testing techniques rather than test automation techniques. Edvardsson in [22] made a survey of test data generation based on code. The paper explains different techniques of automated test data generation with examples but it ignores the explanation of different types of program analysis information. Phil McMinn [24] has surveyed the applications of metaheuristic search techniques such as hill climbing, simulated Annealing, and evolutionary algorithms. The paper shows future directions of search based techniques to structural, functional, non functional, and grey box testing. Bertolino survey [5] focuses on research of automated test data generation techniques that can contribute for 100% automation of testing in industry. Mahmood in his master thesis [10] made a systematic review of Automated Test Data Generation techniques of the period 1997-2006. The paper is good, informative but it ignores technical details of the test data generation techniques. Saha did mapping of different test data generation techniques [9]. The specification based survey can be found in [23].

2.6 Future Challenges

2.6.1 Improvement of code Coverage

Empirical study for code coverage in different existing test data generation algorithm is a key area of research [64, 65, 66]. The computational complexity of test data generation algorithms are very high. Therefore optimal solution or heuristic can be derived for complexity and coverage to facilitate the test automation process in minimum cost.

2.6.2 Efficient Predicate Constraint Modification

Major problems with path oriented testing is that many paths are non executable or infeasible. To decide the feasibility of paths we may generate a set of constraints and then decide their satisfiability. Improvement of constraint solver is a challenging research area till today specially if it contains non linear constraints. Most of the test data generation algorithms complexity is high. It is basically designed for unit testing. But unit testing has high cost for huge quantity of extra coding necessary for simulating the environment where the unit will be run [5]. Therefore the test data generation algorithms can be improved for making them scalable, i.e. for performing system testing.

2.6.3 Loops handling in Path Oriented Testing

Infinite looping is a common error in programs. In fact, it is impossible to detect all kinds of infinite looping fully automatically [53] . But many infinite loops can be detected automatically. Therefore research can be done on early detection of infinite loops automatically and it is more challenging if loops are nested.

2.6.4 Detection of Path Infeasibility at the earliest

One of the most time consuming tasks of automatic test data generation is the detection of infeasible path after execution of many statements. In [3], Korel uses backtracking and path infeasibility may be determined in the last predicate. Work can be done to propose heuristic for infeasible path detection to avoid unnecessary computation. This is a major problem of test data generation based on actual value. In [2], Zhang generates set of constraints by symbolic constraints and then check their satisfiability to determine path feasibility. But the algorithm for solving constraints is complex [2]. In [25] Ronald proposes a method to utilize the best of the previously traversed paths at each selection of a new input data that reduces the computational requirements. More research are required to recognize path feasibility as early as possible to avoid expensive, useless computation.

2.6.5 Loops and Array Program Testing

Test data generation of variable length array and variable number of iteration is a very difficult task. The coverage of test data is highly dependent on coverage of loop, because large amount of time is required for executing in loops. Therefore special attention should be given on loop. It is observed that Path Prefix method [25] is better for program with variable number of loops and arrays. But the problem of this type of programming construct is that the number of path may increases exponentially with the increase of iteration and array size. In [41, 48] given heuristics to predict the minimum number of iteration required for better coverage. But it is difficult to set a common heuristic for all types of programming construct. Therefore a heuristic table for different programming construct may be useful. Empirical studies like [67, 60] should be done on different example programs.

2.6.6 Dynamic Data Structure

Most of the research in testing focus on basic data types such as integer and real numbers. But modern programming languages has constructs with dynamic data structure. In [32] B. Korel proposes goal oriented test data generation involving dynamic data structure via data flow analysis and backtracking. In [34] Viswanathan applied recursive algorithm on constraint simplification to generate test data for dynamic data structure. Zhao in [33] generate test data using least restrictive shape. Finally Sen in [4] generate test data for dynamic data structure program using hybrid approach both concrete and symbolic execution. Research can be done to propose efficient intermediate shape.

2.6.7 Constraint Prioritization

In general, automated test data generation is an unsolvable problem. In symbolic execution we may generate certain constraint which is not solvable specially among the non linear constraints [68]. Xiao in [13] proposes a constraint prioritization method using data sampling scores to generate test data even the set of constraints is unsolvable. The method stores the sample data with their scores to be reused in later constraint solving. Although it can always generate test data but the method may be poor in terms of code coverage and the orthogonal spacing is not applicable for single variable constraints.

2.6.8 Improvement Of Scalability

Due to increase of software size and networking software, testing advocates for non functional measure like performance, scalability and reliability. Performance testing , load Testing, and stress testing are done for determining the scalability of the test data. The need of this types of testing is not to find bugs but to help us in failure for regression testing[69, 70]. Till date there is less work on this type of testing. More work is to be done for these testing in coming days specially for software that run on networking environment. Scalability testing heuristic can be developed for optimal resource utilization to reduce the overall cost.

2.6.9 Test Effectiveness

There are different test data generation techniques which can be implemented using different methods. Irrespective of methods or techniques our ultimate objective is to generate test data which can detect faults. Again faults may be different types. What type of fault is best detected by which method that is the selection of coverage criteria for different types of fault to maximize error detection and also to determine by statistical testing which part of the software is more error prone where we require rigorous testing. Research on code coverage analysis and adapting testing is required to determine effectiveness of test data [71, 72].

2.6.10 UML Based Object Oriented Software Testing

The UML is a set of techniques for specification, visualization and documentation. UML is used by testers for getting greater flexibility. The basic idea is to test software whose design is modeled using UML. The UML based testing is useful for model based system testing of distributed, component based systems. UML sequence diagram, state chart diagrams, UML communication diagram, class diagram, and activity diagram are used for test data generation where we require constraint solving. The main challenge here is to collect information from one or combined UML diagrams and store in a efficient data structure. Test specifications and test data are collected from the data structure.

2.6.11 Agile Testing

The basic principles that form the basis for Agile testing are communication, simplicity, feedback, and iterations. It aims to fulfill customer requirement timely manner [73]. The customer are considered as the part of the project. There should be a close relation between developer and tester. The testers help each other in finding the quick solution. In this approach simple function is taken first and then extra functionalities are added. The agile approach use feedback at every step from customer. To perform agile testing, agile approach to the development is mandatory. Basically in agile approach the entire software is divided in small modules and then identify the priorities of the modules depending on user requirement. The number of iteration should be reduced.

2.7 Trends for Future

In this chapter, we have given a comprehensive on different types of test data generation techniques. The chapter gives an overall idea in the field of automated test data generation. The figure 2.11 shows the automated test data generation methods. Our work may help the readers to select the area and study the concepts of related area. After studying there related area of references the reader may choose one of the future challenge area stated in section 2.6 for research and they may hunt in that area. We have listed the direction of research in a concise manner.

2.7.1 Direction of Research

1. Development of efficient construct solver.
2. Empirical study for finding heuristics for different types of programming constructs to reduce the number of iteration required for test data generation.
3. Loop Bounds and Infeasible path detection for WCET analysis
4. Improvement of scalability of test data generation specially for networking software.
5. Test data generation without path selector. Avoidance of infeasible path during test data generation process.
6. Coverage based on priority of code segment. Prioritization of code with respect to coverage to reduce overall cost.
7. Improvement of test data generation for programming construct having dynamic data structure.
8. Test data generation for programs having variable length array and loops with different dimensions.

9. Test data generation for test effectiveness and code coverage analysis.

10. A comparative study of different test data generation implementation techniques to determine best method for a program.

11. Testing object oriented software using UML modeling.

12. Test data generation for recursive programs and procedures/fuctions.

13. GUI testing.

14. Agile testing.

2.8 Conclusion

Automated test data generation is an important area of research for reducing cost of software development. Test data generation is done to satisfy functional, non functional, and business requirements. Some non functional requirement testing can be done only by automation; where manually it is not possible. The chapter describes four types of architectures. Architecture I and II requires path selector phase. Architecture III and IV avoid path selector phase. The problem of infeasible path can be eliminated only by considering an architecture which has no path selector phase. Architecture I requires constraint generator and constraint solver where as architecture II requires test data generator. Architecture III avoid constraint solver by contributing filter on all paths to collect unique feasible path. Architecture IV is useful for object oriented programming. Depending on test information requirements, test data generation methods, different program analyzers are used. The mapping between program analyzers and test data generators can help in this regard. The chapter emphasizes the basic concepts of automated test data generation. The chapter does not focus much on the test data generation using UML and object oriented methodology. In future, we may hunt the area of object oriented program test data generation works, but without having the concepts of this chapter, we can not go directly to the concepts of object oriented programming. The authors feels these concepts are mandatory to perform research in the area of automated test data generation whether it is conventional programming or modern programming.

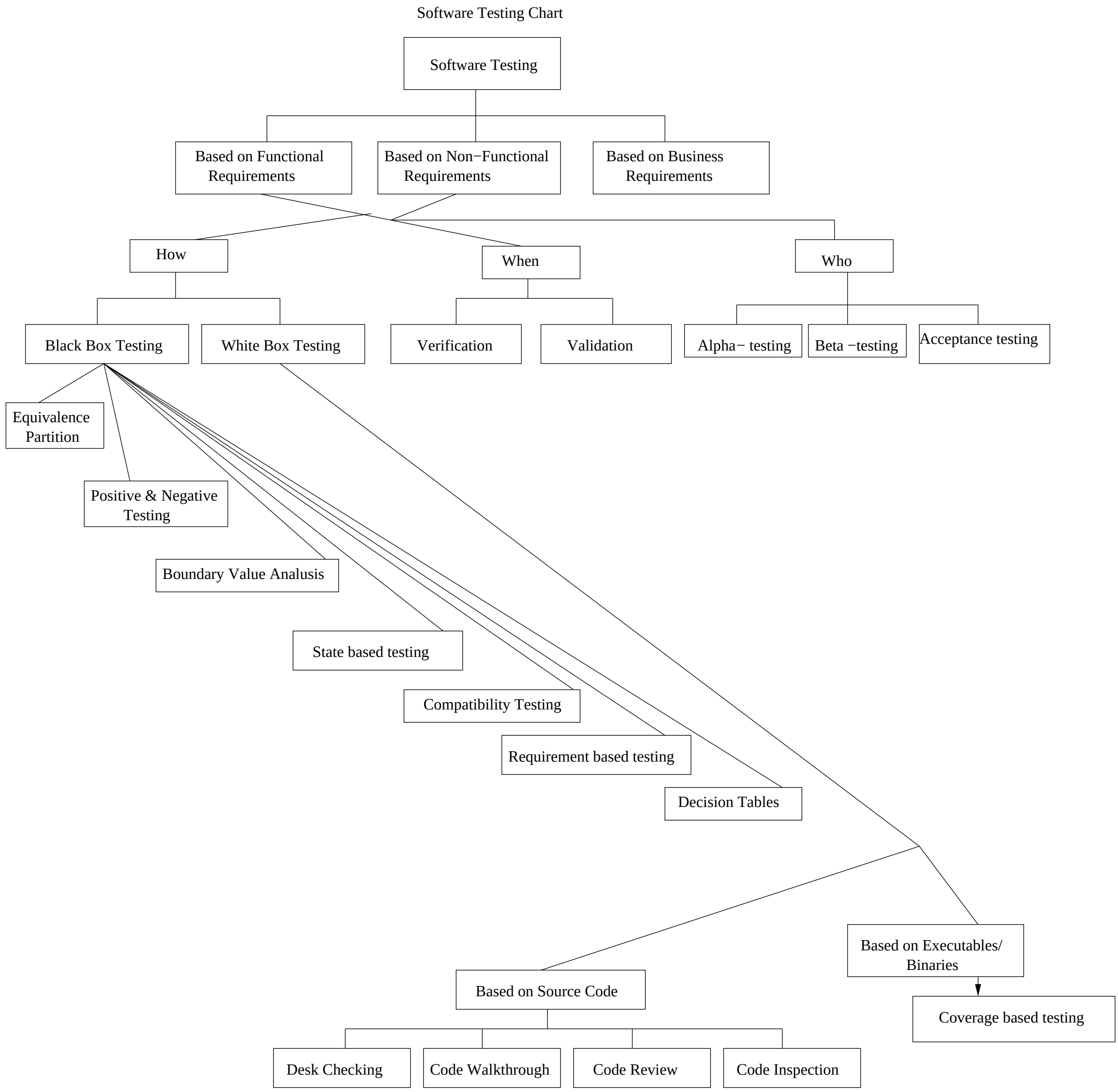

Figure 2.1 : Schematic view of software Testing

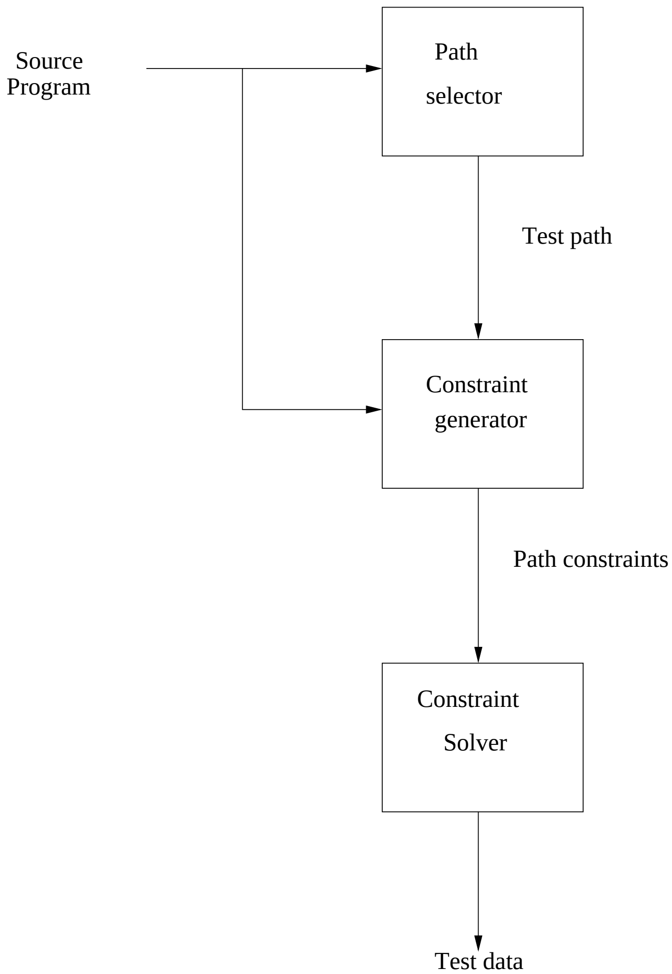

Figure 2.2 : Architecture of test data generation I [16]

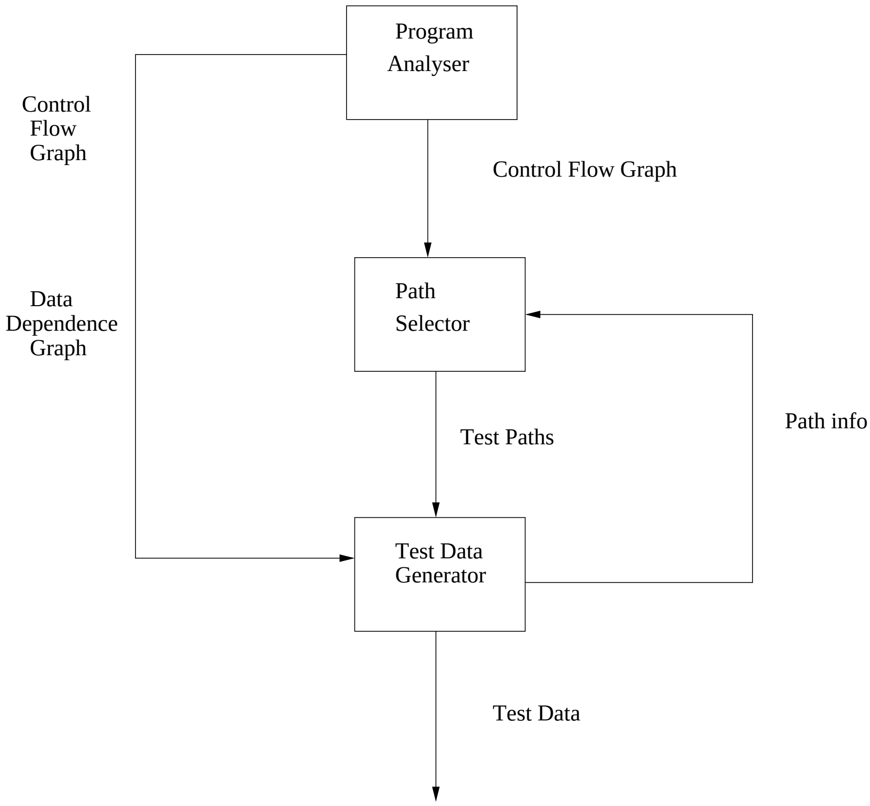

Figure 2.3 : Architecture of test data generation II [3]

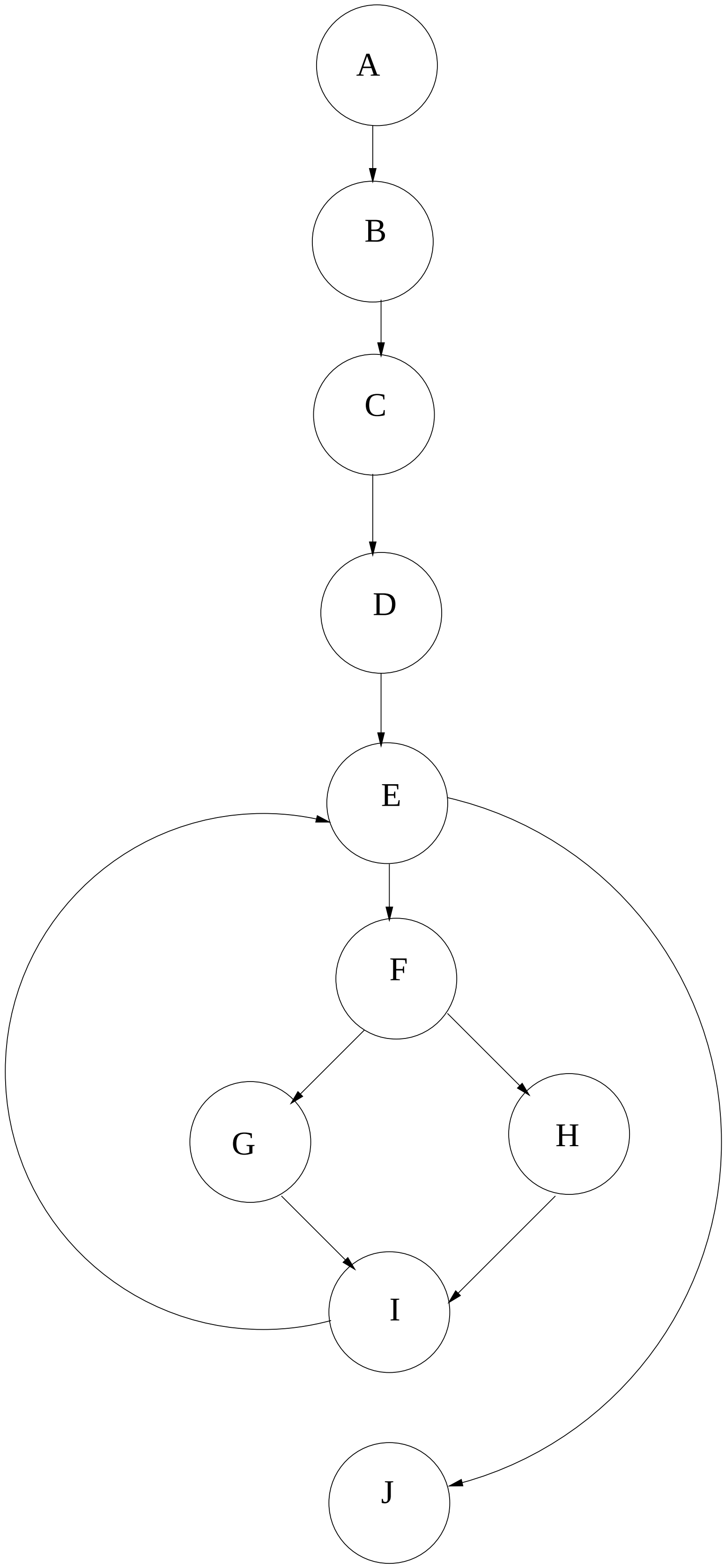

Figure 2.4 : Control flow graph for GCD computation

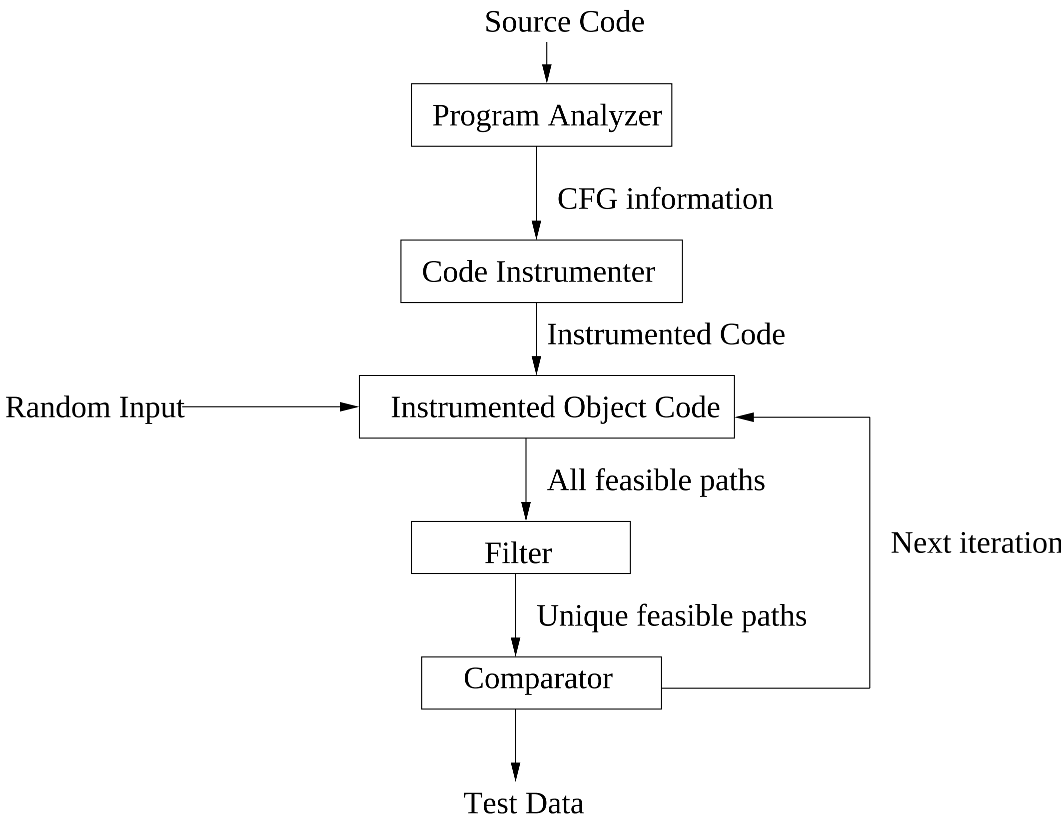

Figure 2.5 : Architecture of test data generation III

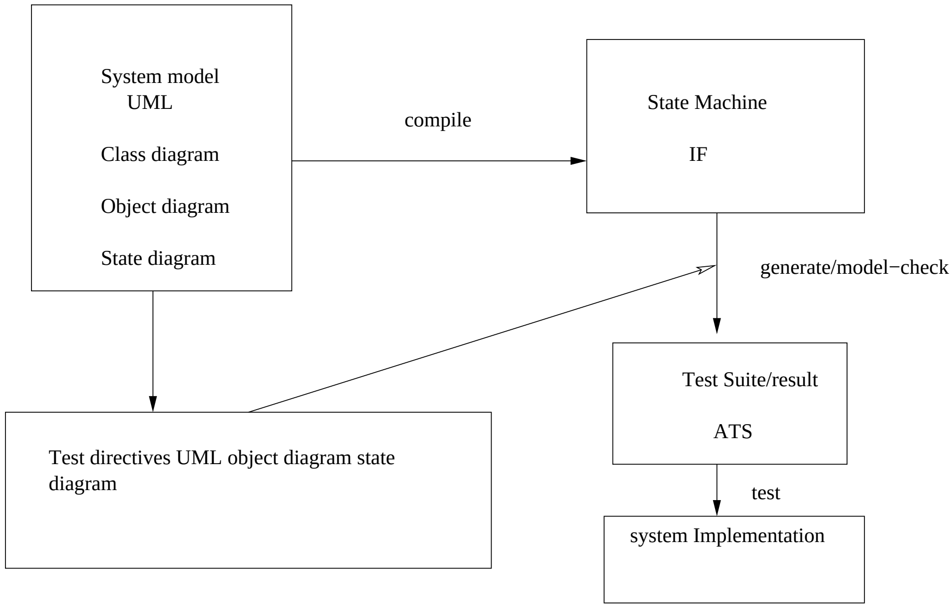

Figure 2.6 : Architecture of test data generation IV [17]

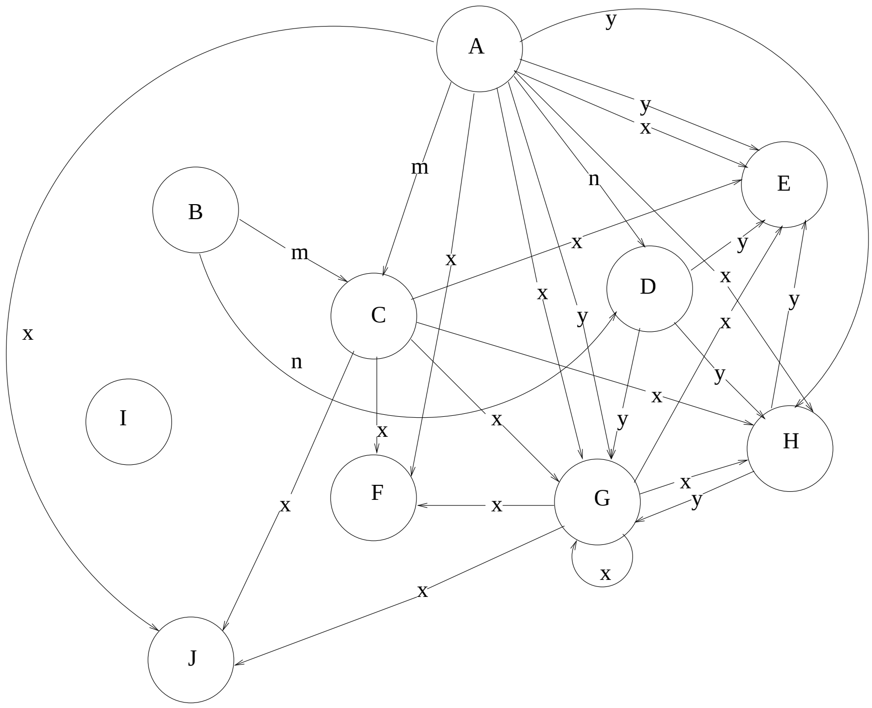

Figure 2.7 : Data dependance graph for GCD computation program

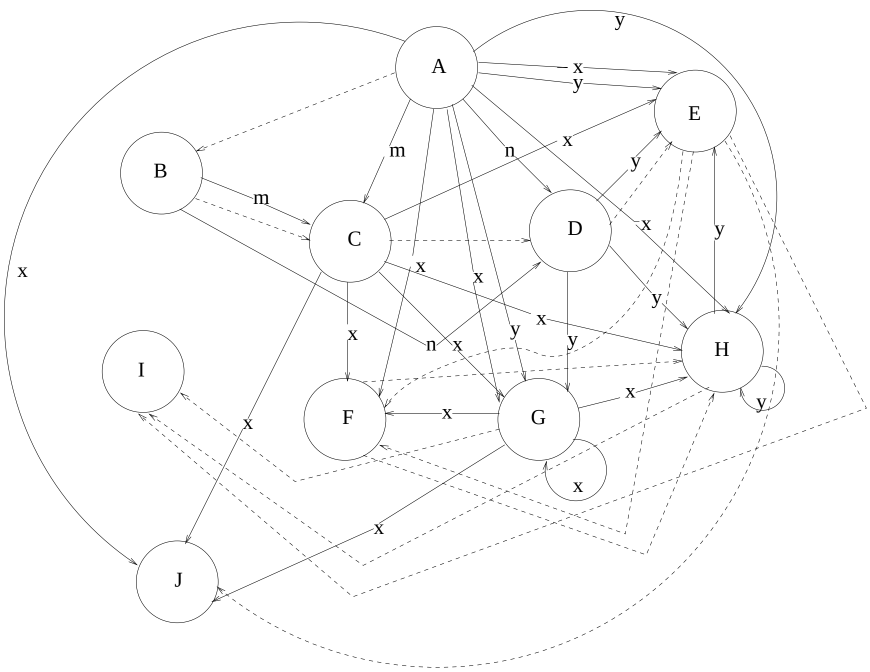

Figure 2.8 : Program dependance graph for GCD computation program

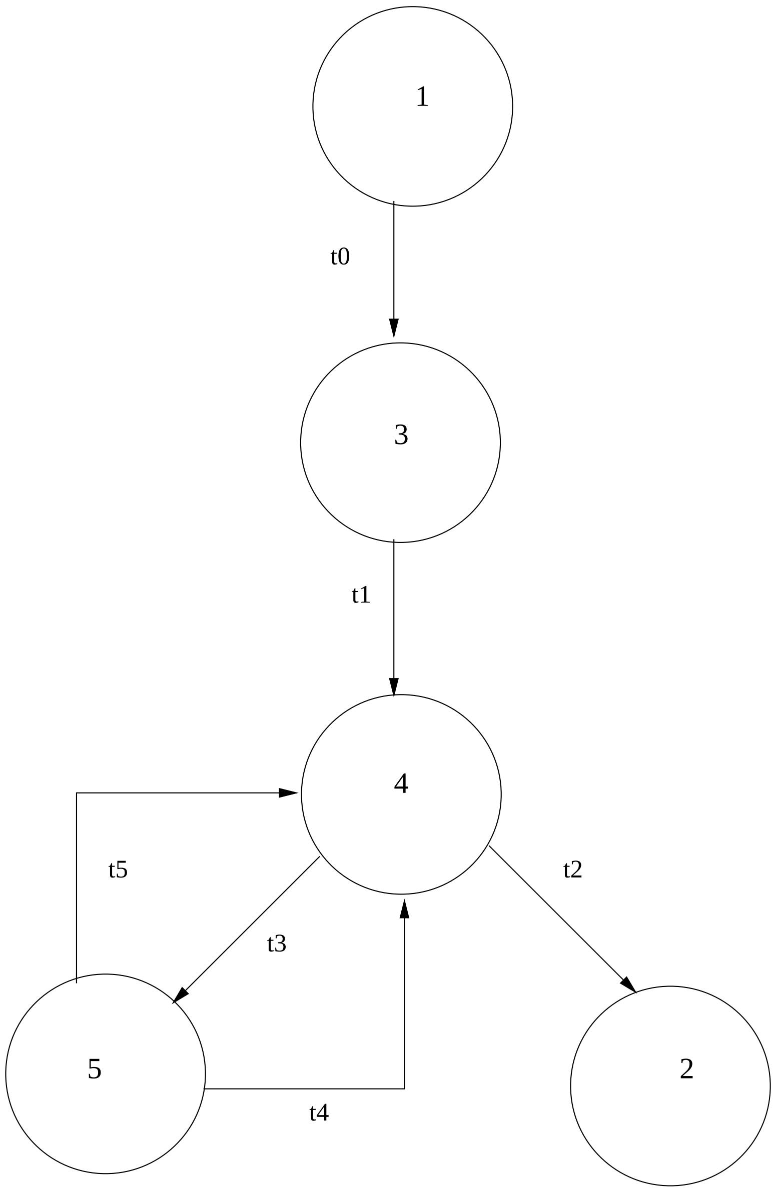

Figure 2.9 : EFSM for gcd program [18]

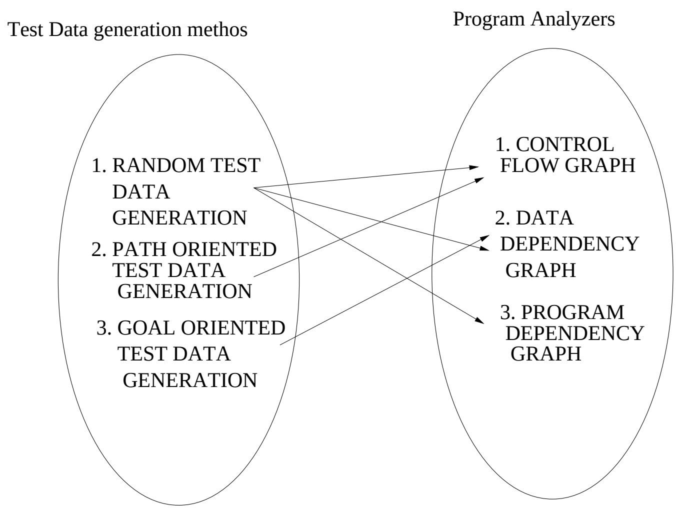

Figure 2.10 : Mapping Between Program Analyzers and test data generation

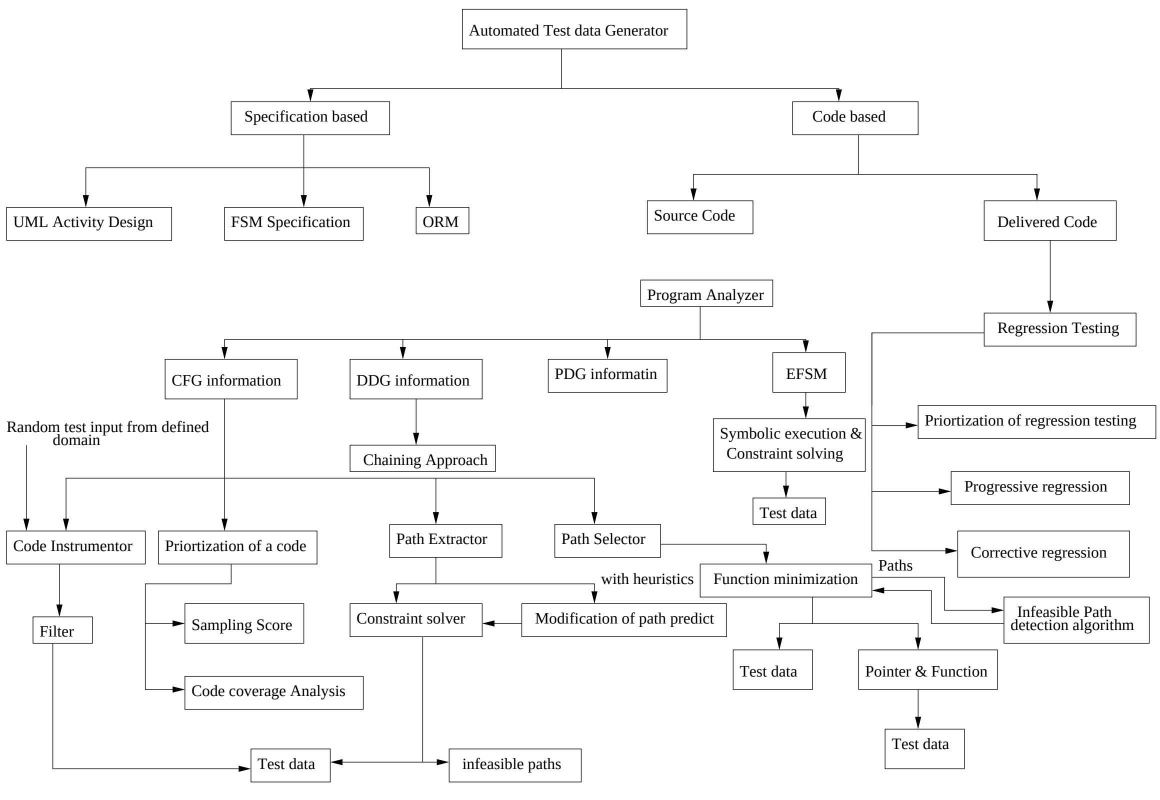

Figure 2.11 : Automated Test Data Generation Methods