Model-based Software Development – A Programmer's savior

Swarupa Ramakrishnan
North Carolina State University
Raleigh, North Carolina, USA
sramakr6@ncsu.edu

ABSTRACT

Programming the most efficient, error-free code for an application involves multiple phases of development, testing and correction. This requires a lot of human labor and machine operations. Finding bugs within a program, introducing exhaustive test cases to check the effectiveness of developed code and comparison of programs to yield the most efficient implementation are just few of the many aspects to application development. Through automated software engineering, most stages of this process have been made much simpler, by segregating the tasks and creating tools for each of them. Throughout the years, there has been a great levels of improvement on this front, with the use of enhanced machine scripts and tools, to automate the laborious tasks involved. This allows the programmers to concentrate on the core of development, without worrying about the edge case parameters and scenarios. This paper briefs about the multiple tools that are in place for this use and highlights the trends ad improvements seen with software engineering with respect to design of efficient code.

Keywords

Code summarization; Symbolic execution; Field failures; Concolic testing; Search-based software testing

1. INTRODUCTION

Software Engineering is applicable to almost everything under the sun, which requires engineering and can be made more effective in terms of development, design or maintenance. It focuses on creating efficient software for making lives simpler. There are major categories into which the domain could be split, including design, testing, tools, evolution and quality management.

Automated software engineering relates to how the above-mentioned engineering tasks can be further automated to ease the usage by target audience. It would not be wrong to extend this phenomenon as 'automating the automated'. The world is gearing towards this idea, in order to achieve effective deployment of all major tasks, lifting labor off humans to the maximum extent. The power of machines to work and optimize these heavy-duty tasks by means of programming and tuning, has been improving by the day.

The most fundamental block of any piece of software is the underlying code. This is not often created overnight, with a single person's hard work. The quality and efficiency of each program defines how the application running over it is crafted and operated. Automated software engineering has shown a great contribution in this particular aspect, of working with the code of multiple programming languages, so as to produce error-free, efficient applications. From autocompleting certain classes and finding bugs, to creating test cases and suggesting possible code corrections, inventions and developments in this front have been nothing but a boon to the application programmers.

Various software scripts and tools have been written throughout the years, in order to strengthen the capabilities of the application code. This is advantageous such that the application is built with all test cases in mind, thereby eradicating possible errors. These tools are created as a one-time program, which can be plugged into numerous other frameworks in the future. Thus, they would have to be compatible with various programming languages in a multitude of possible scenarios. These tools are often not very complex, targeted to a particular operation only. There are some scripts that run on programs, just to find the regions of program that can be further optimized.

Testing phase is greatly benefited from this automated software, yielding highly accurate results within a small frame of time. This lifts the burden off humans to think of all possible scenarios and record the results and performances for each of these inputs. There are many compilers which come with built-in checks for bugs, but it is shown later in the paper that they operate on the overview, not really providing the errors in-detail.

This paper provides an exhaustive literature survey of all similar works that have been done on this front, highlighting the growth and trends of engineering with programs using automated design. It also mentions the drawbacks and possible improvements that can be implemented in the near future.

The sections of this paper are categorized as follows: Section 2 describes the methodologies that are incorporated in this paper, in the case of bug detection and correction, generation of test-cases and selecting a candidate from a frontier of related solutions. Section 3 elaborates about the papers from 2008 till 2010, the proposed models and how they performed. Section 4 talks about the papers that specifically were published in 2011, stating the growth of automated software engineering over the years and the triggers that led it to become what it is today. Section 5 discusses about the papers published after 2011 till date, followed by Section 6, which provides an in-detail trend comparison and developments in the implementations throughout the years. The paper concludes with the Section 7, which also states possible areas for improvement for future work.

2. METHODOLOGIES

In order to analyze and create tools for the various uses described above, it is required to have strong mechanisms and methodologies to scan the program and implement the tool to cater to the need. Depending on the requirement, the tools are developed in a certain manner, in terms of accepting inputs, processing and generating the outputs.

Some of the important methodologies, which are commonly used in most tools, are briefed in this section.

2.1 For Detecting Program Errors

Detection of program errors is probably the base of all other tools that are developed on top of it. It forms the most important aspect of any implemented design, to ensure that the framework runs for any case without getting locked into errors. It is humanly impossible to scan through a program and identify all the possible errors. Compilers do return most of the syntactical and memory-access based errors, but it does not understand the structuring of the program to return logical errors. Various automated scripts and engineering tools have been built for this purpose, especially in the first decade of this millennium.

The basic structure of the error detection framework [1] is depicted in Figure 1 below.

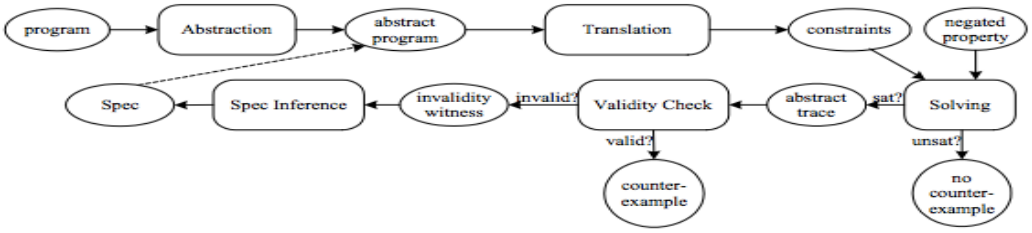

Figure 1. Overview of Error-Detection Framework

The program is initially converted into an abstraction and is translated to check for constraints and specifications. Based on this, possible validity checks are computed and run on the original program. This would yield the results of the different possible executions of the program. Path predicates can be calculated, which are formulae indicating the path of program flow. This ensures that all distinct cases have been iterated through exhaustively.

Errors due to multithreading programs are usually not identified by the simple detection mechanisms. Performing odd edge case runs specifically picks out these errors for structured program flows. The end-to-end semantics of the code is analyzed and the GUI of the application is observed for all possible code branches. This would prompt even the undocumented and previously undetected errors, since it does not detect errors based on similarity with previous data.

The tools can also obtain data from code summarization, which refers to the documentation of the program, explaining the changes made from previous commits. These usually have description about the code modifications, future changes that need to be accounted for, present bugs and test cases that need to be included. In addition to this, differencing of two code versions states the exact lines, which have been altered. These are fed to the tools, which work upon the changed blocks to test the operations of changed code. This is efficient in the sense that the entire program does not need to be tested for each version or commit. It speeds up the detection process and treats each module independently.

2.2 For generating Candidate Fixes

Once the errors are detected, it is required to rectify this without affecting the rest of the code. This can be done manually, which would require the programmer to set aside time to work on the correct solution. A small error can grow into a huge pain, as any solution to it might end up affecting the other classes or dependencies. Fortunately, tools are also created for this purpose using automated software engineering, in order to relieve the burden of spending countless hours on trivial (in most case) errors.

Object behavior models can be plotted for the program, which signify the characterization of the behavior of the code as finite state machines over object and method calls. They are extracted by tracing the program execution and logging their states at different instances of time. These are extremely helpful in identifying efficient fixes for the errors in hand.

Based on successful and failure runs of the reference program, the models of these runs are compared to check what might have triggered the error in a particular scenario. The branch cases are traced out and the cause of error is identified. These are then passed over as inputs to the next phase of the model, which would develop the candidate and validate fixes to rectify the detected bug. Figure 2 shows the flow of this process [2].

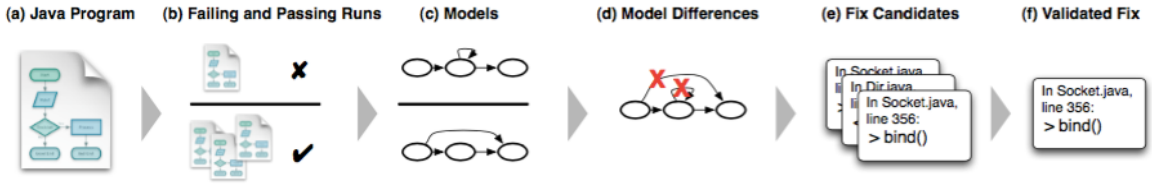

Figure 2. Sequence of generating fixes

Candidate or validated fixes are plugin code snippets are generated though the object behavior model, region of error code and the predicted cause for it.

The generated code snippets are termed as fix candidates and they are plugged in to the program and tested if the errors are eradicated. Usually, a large number of fix candidates are generated, since only a couple of them would eventually prove to

be right for the detected error. It is also ensured that this finalized solution does not disturb other parts of the code. The finalized solutions are referred to as the validated fixes, and result in a successful run of the program when replaced with.

2.3 For Test-Case generation

Programs require efficient and extensive testing in order to ensure the stability and durability of designed code. When done manually, there is always a benefit of the doubt that some cases may be neglected or not looked upon. Each branching or conditional statement in a program leads to a different outcome and in order to test the program exhaustively, it is necessary to account for all these possible scenarios.

Tools have been generated for this purpose, so that one can automate the generation of test cases for a specific program. There are a number of ways in which this can be executed in an efficient manner, such that the reference program is tested of all possible cases in the least amount of time with restricted resources.

Structural analysis of program flow and symbolic executions are ways to generate automated test cases, accounting for all the possible scenarios that may arise during program execution. Symbolic execution refers to the phenomenon by which program codes are interpreted using symbolic measures, instead of concrete measures, to obtain desired outputs. The input variables and expressions are solved using the symbolic representation of values and the computations are performed. Since this does not work with static data, it covers a wide spread of test cases due to dynamic executions of the program.

The test-data generation can also be performed based on the application of a dynamic optimization-based search [3] for the required test data. The same approach can be generalized to solve other test-data generation problems. Three such applications are discussed-boundary value analysis, assertion/run-time exception testing, and component re-use testing.

Another mechanism for test-case generation is by concolic testing. [4]. It is a phenomenon where software verification technique and concrete execution are used to obtain the symbolic execution of code and execution paths of a program. It is used as static, along with an automated portion of code, to generate test cases for the presence of bugs within a framework. The merging of static and dynamic analysis instills flexibility in search and stretches the spread by a large margin.

Depth-first search strategy, based on Control-Flow graph, yields an implicit tree of possible execution paths of the program. Since there may be a large number of possible execution paths, common testing data is bounded such that the search is performed within a limited area and is depth-limited. This ensures that the program does not run for tests endlessly in a loop manner, since most paths are entangled with one another at some point.

The simplest technique used for automating test generations is random testing, which runs on randomly generated inputs and operates in negligible time. However, it does not provide the guarantee that all possible behaviors of a program would be tested.

Other means of testing include the following: [5]

2.3.1 Search-based methods

These are used to scan through the program for all possible cases or scenarios and develop test cases to verify the functioning of them all. Search-based methods are efficient but often get stuck within local optima and degrade when the landscape does not offer guidance.

2.3.2 Constraint-based methods

These constitute the other type of methodology used to generate test cases from a program. The drawback of these methods is that, they are capable of handling only a subset of the domains and is not entirely scalable.

2.3.3 Genetic Algorithm

Genetic algorithms (GAs) evolve a population of candidate solutions, guided by a fitness function, towards achieving a given coverage criterion. They avoid from the search getting stuck by mutations and crossovers. They operate over the program, optimizing on the go to obtain a large spread of possible test cases. It is efficient in the sense that it operates in a ‘smart’ manner, by categorizing the more important and less important fields as the execution occurs.

2.3.4 White-box testing

It refers to the task of generating suitable inputs for programs using the program’s source code. These are mainly driven by intuitiveness and can be done manually. It is termed to be the least efficient among all other methods. The effectiveness of this random testing mechanism can be improved to an extent by means of setting a maximum length of test runs. [6]

2.4 For choosing the most efficient code for a given application

There is never a single way of implementing a particular application or framework. There are always multiple codes and programming decisions that need to be agreed upon, before moving ahead with the final program. This decision is affected by many parameters that have dynamically varying priorities based on the requirements and setting of the application. Based on these conditions, the best program that lives up to all requirements, with the least amount of undesirable properties is set to be the most efficient program for the application.

In certain cases, reducing the energy may be the most important criterion. In certain others, it may be the cost, execution time, or even the number of dependencies or abstraction.

Reducing the energy usage of software is becoming more important in environment, [7] particularly battery powered mobile devices, embedded systems and data centers. Empirical studies indicate that software engineers can support the goal of reducing energy usage by making design and implementation decisions in ways that take into consideration how such decisions impact the energy usage of an application. However, the large number of possible choices and the lack of feedback and information available to software engineers necessitate some form of automated decision-making support.

The best solution is achieved by analyzing the priorities of the participating parameters and tweaking the program accordingly. This can be done manually by programmers, which would result in a lot of effort of time for this trivial issue. This time and resource can instead be used to develop or deploy something else. Thus, automation comes to the rescue in this manner by generating similar solutions and allowing the user to select the most specific solution, according to the needs.

This is done by means of a model [8], which accepts a program as input and analyzes the code to check for the statistics on the

regions that take up more time, memory or resources. Once this is done, the model aims at creating duplicated of the program which each work on optimizing the parameters. This can also be developed as a multi-parameter optimization technique. A list of candidate solutions is produced, which each logically function the same way as the reference program but have reduced speed, or memory or utilization.

The user would just have to choose the most apt solution for the environment that he is currently working in, so as to efficiently operate that in the current scenario. This lifts the burden of programmers and also, delivers the efficient solution to every problem that may arise.

3. IN THE YEARS OF 2008 TO 2010

Trends have shown that the growth in software engineering for program execution has not been limited to one particular methodology. There are various tools that have been developed for the purpose of testing, generation of test cases and for auto-correcting certain obvious errors. All these operations use different insights and methodologies, so as to obtain the final output of an efficient application.

In the years of 2008 to 2010, there was much work done in finding errors in a program and coming up with possible fix solutions based on the nature of the bug. Additionally, generation of automated test cases was popular in order to increase the speed up the testing phase of programs. During these years, more importance was given to deploying new tools and technologies, which make the work easier than to invest in applications that did better. The mantra for this era was to 'create and innovate'.

Since this was the period during which numerous Android applications started getting developed, they were many programs amidst these that did not have efficient code structure. A model [9] that aimed at detecting the errors in Java programs was developed. It was able to detect the errors, which do not get picked up by the traditional compilers.

On compiling, only the overall errors are detected, without looking through all branch cases or into multithreading operations. The developed tool, AndroidRipper, performs a structured check on the developed app using the GUI, pointing out faults that weren't previously documented or detected. Not many errors were documented on Android back then, and this tool helped to look into new uncategorized errors as well.

AndroidRipper would prevent applications from being developed with faulty errors in the underlying code and make the framework stable even in cases of odd edge cases. The program flow structure and semantics are observed using this model by observing the application's GUI and exploits all possible situations that may arise. The authors of the paper believed that this would also reduce the time spent on debugging and would prove as a much better alternative to the then existing standard debugging tool for Android applications.

Detected errors have to be found correction mechanisms, which solve the problem without hampering other parts of code. This was also automated using a tool named PACHIKA [2], which rectifies the detected errors by suggesting a number of candidate fix solutions. These are determined based on the type of error occurred and the scenario in which it develops. The tool scans for any dependencies between parts of the code that lead to the error and document the stages. Suitable changes in the affected code are generated and are tested with the run. They are then categorized into successful or failed solutions, finally producing the validated fix solution for the error in place.

To study and test the proposed model, two program subjects are used, namely the ASPECTJ and RHINO. These have listed crashing bugs, along with the size and number of tests within each program. PACHIKA tool is tested for accuracy by checking the number of bugs it is able to fix with respect to the total number of bugs present. Further, it is also used to test the number of validated fixes that are generated in order to correct the failing run and make it pass at all executions.

The program is executed multiple times to categorize the failing and passing runs. It is then interpreted as an object behavior model and differences are evaluated between them. Based on this, fix candidates are generated that rectify the failing cases, from which the test case that converts the fail run to pass is termed as the validated fix solution.

Removing redundant code in the program is an important feature of any model [10],[11]. Though it is not often used, it can be seen that the later papers incorporate pre-processing stages in the model to make the tool more efficient. Having removed redundant code, API usage can be reduced. This process can be automated so as to achieve easy retrieval of the core program. A technique and tool was developed to support automatically detect patterns of API usage in software projects. The main hypothesis underlying this mechanism was that the client code imitating the behavior of an API method without calling it might not be using the API effectively because it could instead call the method it imitates. The software systems are analyzed to detect cases of API method limitations and this lead to many improvements on the quality of the code.

Other than detecting and correcting the errors within the program, there were many papers that propose tools to automatically generate test cases for the program. Since testing took up a lot of human effort and was not easily performed in an exhaustive manner, it was highly desired to make this process automated.

The basic methods of creating test cases are listed and compared based on their heuristics in [4]. Test case generation is an important element of development, where the developed framework is searched extensively for bugs. This can be automated using multiple existing features, which each use different search algorithm and operate on the inputs in a different manner. Although symbolic and concolic techniques have been effective in achieving this for small programs, they fail to function fast and effectively in larger programs. In these cases, only a tiny fraction of the total number of possible program paths is covered, thereby yielding less accurate results. A couple of search algorithms have been implemented that would cover a significant portion of the branches in a test program fast, using limited amount of space.

The search algorithms developed are CFG - directed search, uniform random testing and bounded depth-first search. In the first algorithm, a static structure of the program is plotted with all possible branches denoting execution paths. These branches are chosen to negate the purpose of test generation based on the distance in CFG to the uncovered branches. This greedy approach ensures that the coverage area is maximized so that the search can be performed faster and more efficiently. In the second algorithm, the program space is sampled uniformly of all possible program paths and then performed the traditional random search. This

controls the randomness of the technique, resulting in favorable outcomes. The last algorithm limits the number of branches travelled to an optimal depth such that maximum coverage is obtained from minimal space.

Symbolic Execution of Java Byte code is another mechanism by which test cases can be generated. Symbolic execution refers to the phenomenon by which program codes are interpreted using symbolic measures, instead of concrete measures, to obtain desired outputs. The input variables and expressions are solved using the symbolic representation of values and the computations are performed. In [12], the authors of the paper propose the Symbolic PathFinder tool, which automates performing the symbolic execution of Java byte code. It works on various input data types and carries out operations between these variables with polymorphic class hierarchy. SPF handles programs with parallelism enabled, as well as a combination of concrete and symbolic execution platforms.

Certain Java programs may have unspecified inputs, thereby creating an absence of concrete values that can be used for testing purposes. Instead, the Java byte code can be operated on using symbolic executions, replacing the expressions with equivalent symbolic code and deducing the required outputs and results. This would then yield to the possibility of generating automatic test cases for the program, without mentioning any standard inputs. SPF is capable of performing this interpretation and test case generation for the required code logic. This is successfully used in NASA, Academia and industry to detect faulty bugs in code and also ease the operation of clean code development.

Using SPF, the error types of Race conditions or Deadlocks are detected with the usage of minimum resources, as listed in the table below. The time taken to analyze the given program, with symbolic executions and automatic test case generations is minimal and thus, is shown to be very efficient. Various Java class libraries are operated upon using SPF, in a systematic procedure of multiple operations. The program is tested for instructions, branching conditions, decision constraints, input data structures, etc., to analyze the different aspects of the program and deduce the final report.

To depict the efficiency of the tool, it is combined with the Java PathFinder verification tool-set where the JPF-core is replaced by the SPF to employ symbolic instead of concrete executions. User specifies a depth to which the program is to be analyzed so as to make sure that the tool does not endlessly execute the symbolic code. The tool is run on multiple Java classes to test if the subtle bugs are uncovered.

4. THE YEAR OF 2011

This year saw many solid innovative ideas, which led to the improvements in Automated Software Engineering, as a whole. In the areas of automatic test case generation, there have been improvements such that better methods are used for implementation, thereby resulting in more efficient test candidates. The spread of generation is larger and accounts for all the possible edge cases that may arise in a program. Genetic algorithms are introduced, which is a mutant of search-based and constraint-based methods for test creation [13].

Though each of these two predominant methods work effectively and produce reasonable results, they have a number of drawbacks. In the case of search-based testing, efficient algorithms are applied to find inputs that can best serve as suitable tests. Though this can handle any code and scales well, it depends largely on the type of guidance provided by the program landscape. If a flat program or a program with multiple local minima/maxima is given, this method fails to work as effectively. In the case of constraint-based testing, static and symbolic execution is used to precisely calculate the inputs. Though it accounts for search dependent on heuristics, it does not prove to be scalable and depends on the type of constraints set. Thus, the idea of combining both methods, exploiting the advantages of both, yields a hybrid search GA.

An initial population of candidate solutions is to be generated randomly for the program under test, using both search methods. Individuals are then crossed over with a certain probability and mutated using standard mutation operators. The path conditions for the considered individual are collected using Design Space Exploration (DSE) [14], following which a condition is selected randomly. A new constraint system is created, consisting of the path conditions that lead to the selected branch, conjoined with the negation of the selected path condition. This constraint set is passed to a constraint solver, and a solution to the constraint system represents a mutated individual that follows a different execution path than the original individual. If the constraint solver fails to find a satisfying assignment, then standard mutation is used as fallback. After determining the fitness values of the new population, the GA continues iterating until either a solution has been found or some other stopping condition is met. This overcomes the individual problems of search-based testing and constraint-based testing and proves to be more efficient than either of them alone.

Running GA-DSE, GA, DSE, did testing of the proposed hybrid model and random search on each of a set of 20 case study subjects as listed in the table below. The examples consist mainly of linear constraints; Gammq, Fisher, Bessj, Expint, and ASW have constraints involving floating point numbers, as well as some non-linear constraints involving a square root, logarithm, bit-operations, and a quadratic function. The superiority of GA-DSE over random search and the GA is striking, with GA-DSE achieving higher coverage on all examples but Fisher and ASW, where it achieves the same coverage. On the whole, this confirms the expectation that by combining GA with DSE in the GA-DSE algorithm one achieves significantly higher coverage than its constituents GA and DSE as well as random testing.

Another idea that was not closely related to the model-based software development, but could be used to find certain special targeted regions of code. [15] addresses how keywords can be used by a model to obtain more relevant features from a block of text. Normalized Discounted Cumulative Gain (NDCG) metric is used to rank quality by measuring the performance of a recommendation system based on the graded relevance of the entities used. It ranges from 0.0 to 1.0, with 1.0 representing the ideal ranking. It is commonly used in information retrieval tools and to evaluate the performance of programs.

An automatic technique for generating maintainable regression unit tests for programs is presented in [16]. This removes the inadequateness of the then existing test-case generation techniques. They were designed keeping the target feature in mind, whether it was for an application or for evaluating libraries. The test suites that they generated were brittle and hard to understand, and catered to be able to locate bugs within a program. A suite of techniques is proposed that enhance an existing unit test generation system.

Randoop tool, an automated tool that generates unit tests, implements a technique called feedback-directed random testing [17]. In feedback-directed random test generation, a test is built up iteratively and in each iteration, a method or constructor is randomly selected to invoke, using previously computed values as inputs. The key idea of feedback-directed testing is to execute each test as soon as it is generated, and to use information gathered from that execution to bias the test generation process as it creates further tests. The bias makes the test generator less likely to generate illegal tests, and less likely to generate redundant tests. In particular, inputs that create redundant or illegal states are never extended, which has the effect of pruning the search space. Figure 3 depicts the sample working of the Randoop tool, in extending the given parameters to possible configurations.

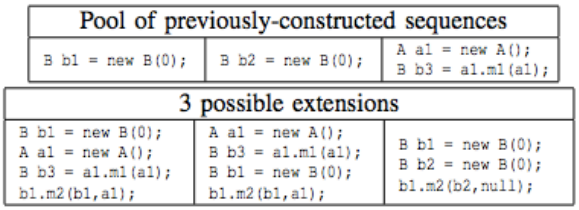

Figure 3. Sample functioning of Randoop tool

5. FROM 2012 TILL DATE

As we move through the years, Automated Software Engineering has vividly gown to include many more creative and efficient technologies and tools in order to simplify the daily operations. In terms of programming, the shift has been made to maintain the current developments of automating tools, and also to add more efficiency to the results obtained. Thus, the mantra of this time frame can be termed as to 'maintain existing creation and incorporate more efficiency'.

The tools developed during this period not only automate the laborious tasks of code implementation but also look more into choosing the best candidates and more efficient solutions. The models must possess the intelligence to do all the things that the previous models did, only much better. They should also be able to comprehend the current scenario and choose the most suitable solution correspondingly.

Comparing with the similar genre of test-case generation, the tools developed during this time were a combination of multiple methodologies, instead of applying a single mechanism. Additionally, the models worked even for the classes with environment dependencies [18]. The environment refers to the file system, network or user-interactions that the classes of a program interact with. These may cause a problem in the case of automatic test-case generation, since the environment may hinder the execution of a process or mess up dependent executions of one program flow.

EVOSUITE, a Java test generation tool, satisfies a coverage criterion, and also includes effective assertions for programs summarizing the behavior of the current execution. The tool is also extended in order to let it accommodate test-case generation of programs in which environment-dependent classes are present. Mocking mechanisms are used to isolate classes form their dependencies. Replacing classes during the testing phase, instead of retaining the original classes, does this. The behavior of this replaced class can be predetermined and programmed in favor of the developer.

Object-oriented software consists of user-level classes, which depend on the environments of the program. Generating automatic test cases for such programs results in a number of problems, since the program flow does not remain static and becomes unpredictable with changes in the environment. One problem is that, models working on such problem so not yield maximum coverage as it becomes impossible to cover all cases by mere function and class calls. If a particular class depends on the contents of a file, it is not possible to traverse this case without turning to the hardware support. Another problem that arises is that even if the class can be covered within the test case, the resulting tests may be unstable. If a particular class takes the system time as input, it may pass for that one time but would fail for repeated trials. Thus, it is important that for efficient test-case generation, the environment-dependent classes must be separated from the remaining program before processing.

Controlling the environment removes unstable tests for intended sources of non-determinism. For most cases, resetting the static state or mocking non-deterministic JVM calls removes all unstable tests. However, there are still a number of cases, which prove to be the corner cases and still require further engineering to rectify the instability in them. The implementation covers a large share of I/O, but fully covering the Java standard library is ongoing engineering effort. Since not all Java API classes are mocked, some CUTs may access the each file system even when EVOSUITE runs on a virtual file system. When this occurs, the default sandbox will prevent harmful operations, but only a fraction of the I/O issues are solved.

In terms of web applications, [19] introduces an important tool, SWAT, for automated testing using Search Based Software Testing (SBST). The algorithms discussed in this paper greatly increase the efficiency and effectiveness of traditional search based techniques discussed before. Since this incorporates dynamic search mechanisms, it yields increased branch coverage and reduction in test effort. The improvements are compared and evaluates separately with those produced by the earlier models having traditional search based methods for generating test cases. These algorithms work better in multiple constraint scenarios, and crashes.

Moving on outside the range of test-case generation, there have been implementations of model which re-create field failures in order to prevent the occurrences of possible errors. Field failures are the faults that occur on user machines due to an unexpected edge case error after the deployment of the software code. It is a major concern for the modern software systems to duplicate the errors in-house and fixing it before the next release.

This is extremely advantageous as most often than not, when bugs are reported, it takes a lot of time to be able to find the fix solutions for it. Even before the analysis of the solution can be performed, it is important to replicate the fault in the field, so as to observe the root cause and possible dependencies. Just be the problem statement, it is not possible for one to come up with the solution to a problem.

Field failures arise even after thorough testing of a software and deployment into the market. This requires in-house bug fixing which involves duplication of the same experienced failure. Since software code can be very complex and there may be numerous paths or cases to a result, it becomes difficult to navigate through

the code in the same manner to produce the desired error. Additionally, the problem faced is usually given as a grammatical input to the software engineers, which does not always provide complete information. Thus, a model has been proposed, Search Based Failure Reproduction (SBFR) [20], to predict the trajectory of function calls or a list of operations that can be performed in order to obtain an analogous error to the one reported. This is performed by means of having the software code, input grammatical description of the fault and sample execution data.

Grammatical Evolution is a type of Grammar Guided Genetic Programming (GGGP) where the individual candidates are sentences, which are formed by following the accepted rules of grammar. GE follows the Context Free Grammar (CFG), which is maintained at all times during mutation or crossover. Following the given steps for this particular grammar can produce the initial candidate set. Genotype - Phenotype is a pair used to represent an individual in GE. Genotype is the encoded form of a candidate that can be involved as such in the search algorithm, while phenotype is the executable version of the candidate that is predominantly used as an input for software under test. The mapping between these two values is used to convert the linear representation into a syntax tree.

The input to the model is provided as a grammatical sentence and needs to be broken down to a tree of genotype-phenotype pairs in order to make it representable to the machine/program. This is performed by scanning through the input and picking out the words and operations, as per the selected grammar. The linear grammar representations can be encoded as a sequence of integers, called codons. Once the input has been made representable, the evolution operators of mutation and crossover are decided. Sub-tree crossover and mutation is frequently used to populate generations of suitable candidates. On passing these generated inputs to the SBFR model, it would yield results in the form of a list of operations which when carried out in order would give rise to a similar failure in the software under test. Being more efficient in computation and ease of operation, field failures should no longer be a concern with the help of this predictive model.

Given an input program P, an execution E that results in failure F and set of execution data D, the model predicts an execution E' that results in a failure F' which is analogous to F. E' also generates the same data D to ensure that the failure occurs in the same manner. E' should also be an actual execution of P and should result in F' without any other additional information. This is illustrated in Figure 4 below.

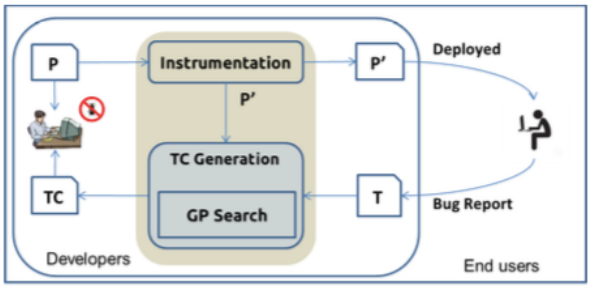

Figure 4. Illustration of the working of model - SBFR

Software Under Test (SUT) is the program P and the result E' obtained is a trajectory that depicts the order of function calls. SBFR uses 80/20 rule, where a probability of 0.2 is distributed normally over the recursive productions and the remaining 0.8 on non-recursive productions. The best solution is found by means of fitness functions, which are applied to the candidate solution to obtain a performance metric. The solutions are then ranked based on this score to obtain the best solution as the output of the model.

Introducing Genetic Algorithms and Genetic Programming to this front has noticed a remarkable growth and production of very efficient solutions. So far, we have seen predictive models majorly for bug retrieval, test-case generation and producing fixes for the faults. However, these solutions have not been accounted for the difference in regard to the resources consumed. There has been no distinction among the solutions, as to whether they consume less memory, or have a fast execution time, etc.

A frontier of possible solutions, which are logically the same as the reference program can be created, where each version has an optimized resource [21]. This keynote paper proposed a model which guaranteed to find better programs using a pareto program surface using genetic programming.

Properties of a program can be classified into functional and non-functional properties. Functional properties include the logical structure and working of a particular model while the non-functional properties include speed, size, throughput, power consumption and bandwidth. Programmers implement programs, keeping the functional properties in mind and later tweak the implementation to better the non-functional aspects of it. This can prove to be extremely demanding to be catered to. Inspired by the results from Search Based Software Engineering (SBSE), this model was devised to generate a set of candidate program implementations, given an input program that satisfies the desired functional requirements. These programs would all share functionality, but each differs in their non-functional trade offs. The software designer navigates this diverse Pareto surface of candidate implementations, gaining insight into the trade offs and selecting solutions for different platforms and environments. This prevents the designers and developers to focus on the edge cases of non-functional properties, which can often comprise of complex inter-related tradeoffs. They are now free to implement more on the core functionality of the models.

The model (GISMOE) architecture is depicted in Figure 5. The choice of platform and fixed software constitute the environment in which the nonfunctional properties are evaluated. The pareto program surface is created by the non-functional evaluators and GISMOE seeks optimal solutions based on these generated candidate solutions. The programmer can choose any of the solution, or combine multiple solutions as a set, based on the requirements. The solutions are compared based on their fitness values. This is fed as an input to the GP engine for sensitivity analysis. Test cases can come from any test data generation technique, for which there are many already available options that can be used as a component to GISMOE. Sensitivity information can be pre-computed before the GP improvement process commences. The sensitivity of the program to each non-functional property is computed using the non-functional evaluation harness. This process requires no knowledge of the functional test cases, since it seeks to identify those parts of the program that are non-functionally sensitive, irrespective of functional properties. The result of the sensitivity analysis is a prioritized list of program elements for each non-functional property of interest.

This model stands as the step to many potential goals and benefits in future. Based on such an idea, similar models can be created for various uses. This is the closest representation of a human mind,

having its capabilities implemented as a machine. The tedious task of tweaking and verifying the runs at each phase, in order to cater to the non-functional needs, is no longer a worry for the programmer as it has all been automated. It is costly, inefficient and wasteful of human ingenuity to let the smartest individuals construct programs on a Pareto surface, when an automated tool can generate a multitude of solutions, which each outperform the human-generated solutions, with a fraction of the time and cost.

6. STUDY OF GROWTH

After reviewing the implementations of model-based software development using automated software engineering, a clear trend of the improvisations and hurdles can now be addressed in detail. This paper has put together the major developments of tools and technologies that occurred in the time frame from 2008 onwards, which are used to make code writing and debugging much more efficient. All applications require base software and this is an important aspect, which if made most efficient, would result in the betterment of the application as a whole.

The early papers proposed models to solve the, what now may seem as, relatively smaller issues in code building. They focused on creating clean structured programs, without any hidden errors that go undetected by the compilers. They involved the usage of search-based or dynamic mechanisms, which scan through the program for possible end conditions. Using this, the cases are built and execution errors are detected. These models also account for threaded operations, multi-hierarchical and layered program flow. The implemented models did not account for the complexities of the program under test. They performed simple sequential scan of the program, to identify the affected or critical regions and work on them alone. Some models based bug finding using differencing operations, which only accounts for the number of lines of code. The algorithms developed for such problems cannot be termed as the most efficient ones.

To obtain the test cases, the search algorithms were made to run directly on the program, without involving extensive pre-processing of data. This may result in many false matches and inaccurate solutions. As we see in the later papers, improvements are made on the search algorithms such that the programs with larger number of lines initially undergo processing before being fed as input to the model. They are stripped off the redundant code parts to obtain the crux or main operations. This results in a faster run and test cases can be obtained in an accurate manner.

When deciding upon the correct solution for a particular situation, there may a number of options from which the model would have to make the best decision. The earlier papers did not include adaptive generation of candidate fixes. Since no optimization or improvisation as done on the obtained output, the model worked on returning the best solution seen thus far. When multiple equally good candidates are present, it would just pick one randomly. The later models adopted adaptive fix generation. Having a large set of possible fixes, the generation of candidate fixes if further mutated for bug fixing based on previous occurrences. Additionally, having a mechanism to limit the amounts of testing and return the solution with the largest fitness metric assesses the impact of fixes. This is performed in Depth-based search of a program, where the depth is assigned to the model, thereby mentioning when to stop traversing the branches so as to avoid loops in program flow.

It is important that before a method is called, it is determined that all preconditions are satisfied. This is another noticeable change in the implementation of model through the years. The minute detail is accounted for the recent tools, for easy filtration of data, instead of tracing the program for every case. The necessary assertions are simply inferred by means of a language, instead of interpreting it from them model. The model is usually complex and reading from it would require much time and processing.

The detection of faults is performed by observing the GUI model of the tree for the application and traversing through all possible branches that can be accommodated. The test cases are generated based on this tree and usually, the generation of this tree takes a long time. This depends on the complexity of the code, length of the code and number of possible end cases. This process can be speeded up by having relevant inputs to the model phase that generates this tree. Instead of passing the entire code, later models include a lot of pre-processing stages so as to decrease the amount of code content that is fed. This causes the tree to be generated faster, which in turn leads to faster test case generation. The older models cater to the bugs and fixes, which are originated by actions, tasks, events or exploration criteria. This is later expanded in the newer models to include consideration of code exceptions during parallel operations, non-structured program flow and account for activity-based interactive applications.

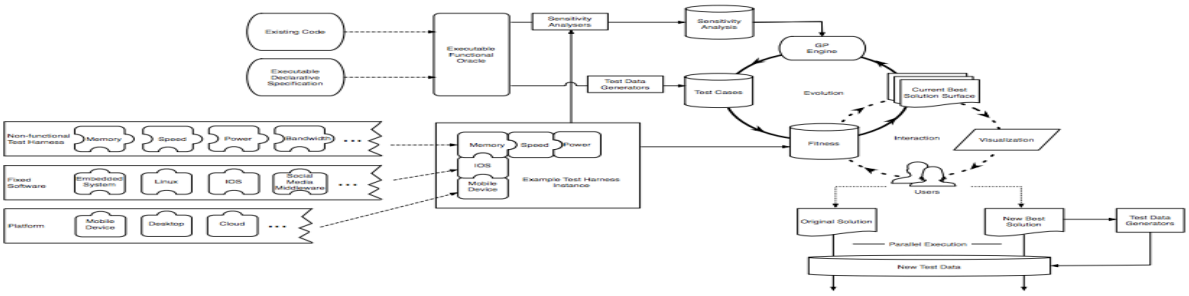

Figure 5. The GISMOE Architecture

Another major issue with the earlier models is that they are not entirely designed to scale. They may analyze the programs in an optimal manner, but as the load increases, the tool tends to show undesired variations in the result. Multiple input constraints cause the model to break. When compared to these models, the implementations in the recent couple of years have been much more effective. They are designed to take up a larger amount of data and input programs, and show effective solutions even in

such cases. However, there is still room for improvement on this front, since the programs are just getting bigger and the models, more complex.

Parallelism of operations is employed in a couple of newer models so as to increase the processing time and efficiency. Since tool can get really complex, each operation takes quite a lot of time. If done in a serial manner, they would take an eternity to complete. Many models incorporated parallel processing at an early state, and the later models have almost all operations pipelined in parallel. The dependencies must be correctly accounted for so as to make sure that the phase output of a model enters the phase of next in a correct manner. The overall code can be segmented into multiple blocks of code to better utilize the parallelism.

Random testing is an approach, which is unpredictable and should not be relied upon to get the exact solutions for a program. Though this method may be the easiest of implementation, yielding the best results in almost all cases. It still follows that certain crucial test cases may be jumped over. Especially in the programs that look into perfect error-free executions for all possible combinational inputs, it is required to have multiple methodologies of testing operated on it. Models that rely only on random testing are long-gone, since many new methods have now come up for this purpose. They include a combinational set of testing mechanisms, so as to achieve complete assurance of the working of he program. As the years have passed, progressions have been made in the invention of multiple mechanisms to obtain the test cases as this always proves to be an integral part of the development phase.

7. CONCLUSION & FUTURE WORK

This paper provides an overall literature review about some about the best ideas that have been proposed over the years in the relevant area of implementing software using automated engineering. It describes many situations, the methodologies used and the outcomes of the predictive model. The improvements made over the years have been studied and problems faced due to different issues are presented with possible solutions.

Based on the literature review, there are a few areas in which there have not been improvements as expected, over the years. This could be a potential area for investment of resources, so as to achieve the expected results that were not really catered to in the models discussed above.

Most designed models use the fitness function as a means of deciding the better solution of the lot, and coming up with the candidate frontier, if required. Thus, the fitness function metric plays an important role in deciding the stake of model. However, the required amount of importance has not been given to these functions in models, so as to make sure that a good solution is in fact very good, and that a bad solution is in fact, bad. Increasing the efficiency of the mechanisms by which the functions are computed would make a significant change in the efficiency of the model. These metrics can also be chosen in a rigid manner, so as to limit the number of candidate solutions being generated. The probability of failure reproduction must be made as high as possible, to eradiate all poor performing programs or test cases.

Another area of improvement is the ability to cater to complex programs in scalable program design. Current model can be extended to cater to complex programs such that the failure scenario can be replicated as efficiently as for the simple code programs. Since complicated software would give rise to more execution data and cases, the models should be tested to be accurate such that the trajectory of function calls yields analogous failure.

REFERENCES

[1] Mana Taghdiri and Daniel Jackson. 2007. Inferring specifications to detect errors in code. ASE 14(1): 87-121.

[2] Valentin Dallmeier, Andreas Zeller, Bertrand Meyer. 2009. Generating Fixes from Object Behavior Anomalies. ASE ’09 *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*: 550-554

[3] Tracey N., Clark J., Mander K., McDermin, John. 1998. An automated framework for structural test-case generation. ASE ’98 *Proceedings of the 1998 13rd IEEE/ACM International Conference on Automated Software Engineering*: 285-288

[4] Jacob Burnim, Koushik Sen. 2008. Heuristics for Scalable Dynamic Test Generation. ASE ’08 *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*: 443-446

[5] Jan Malburg and Gordon Fraser. 2011. Combining Search-based and Constraint-based Testing. ASE ’11 *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*: 436-439

[6] Andrews J.H, Groce A, Weston M, Ru-Gang Xu. 2008. Random Test Run Length and Effectiveness. ASE ’08 *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*: 19-28

[7] Irene Manotas, Lori Pollock and James Clause. 2014. SEEDS: a software engineer’s energy-optimization decision support framework. *ICSE 2014 Proceedings of the 36th International Conference on Software Engineering*: 503-514

[8] Raymond P.L. Buse and Westley Weimer. The University of Virginia. 2010. Automatically Documenting Program Changes. ASE ’10 *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*: 33-42

[9] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine and Atif M. Memon. 2012. Using GUI Ripping for Automated Testing of Android Applications. ASE ’12 *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*: 258-261

[10] Kawrykow, D. Robillard, M.P. 2009. Improving API Usage through Automatic Detection of Redundant Code. ASE ’09 *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*: 111 – 122

[11] Shivaji S, Whitehead E.J, Akella R, Sunghun Kim. 2009. Reducing Features to Improve Bug Prediction. ASE ’09 *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*: 600-604

[12] Corina S. Pasareanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic Execution of Java Bytecode. ASE ’10 *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*: 179-180

[13] Jan Malburg and Gordon Fraser. 2011. Combining Search-based and Constraint-based Testing. ASE ’11 *Proceedings of*

the 2011 26th IEEE/ACM International Conference on Automated Software Engineering: 436-439

[14] Xusheng Xiao, Sihan Li, Tao Xie, Tillmann N. 2013. Characteristic studies of loop problems for structural test generation via symbolic execution. ASE '13 *Proceedings of the 2013 IEEE/ACM 28th International Conference on Automated Software Engineering*: 246-256

[15] Swapna Gottipati, David Lo, and Jing Jiang. 2011. Finding Relevant Answers in Software Forums. ASE '11 *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*: 323-332

[16] Robinson B., Ernst M.D., Perkins J.H., Augustine V., Nuo Li. 2011. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. ASE '11 *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*: 23-32

[17] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In ECOOP, pages 504–527, July 2005.

[18] Andrea Arcuri, Gordon Fraser, Juan Pablo Galeotti. 2014. Automated Unit Test Generation for Classes with Environment Dependencies. ASE '14 *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering*: 79-90

[19] Alshahwan N, Harman M. 2011. Automated web application testing using search based software engineering. ASE'11 *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*: 3 - 12

[20] Fitsum Meshesha Kifetew, Wei Jin, Roberto Tiella, Alessandro Orso, Paolo Tonella. 2013. SBFR: A Search Based Approach for Reproducing Failures of Programs with Grammar Based Input. ASE '13 *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*: 604-609

[21] Mark Harman, William B. Langdon, Yue Jia, David R. White, Andrea, Arcuri, John A. Clark. 2012. The GISMOE challenge: constructing the pareto program surface using genetic programming to find better programs. ASE '12 *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*: 1-14