Genetic Programming Testing Model

Nada M. A. Al Sallami

Abstract— **Software testing requires the use of a model to guide such efforts as test selection and test verification. In this case, testers are performing model-based testing. This paper introduces model-based testing and discusses its tasks in general terms with proposed finite state models. These FSMs depend on software's semantic rather than its structure, , it use input-output specification and trajectory information to evolve and test general software. Finally, we close with a discussion of how our model-based testing can be used with genetic programming test generator.**

***Index Terms*— Model-Based testing; Genetic Programming Test Generator; Finite State Machine**.

I. Introduction

Software behavior can be described in terms of the input sequences accepted by the system, the actions, conditions, and output logic, or the flow of data through the application's modules and routines. In order for a model to be useful for groups of testers and for multiple testing tasks, it needs to be taken out of the mind of those who understand what the software is supposed to accomplish and written down in an easily understandable form. It is also generally preferable that a model be as formal as it is practical. With these properties, the model becomes a shareable, reusable, precise description of the system under test. There are numerous such models, and each describes different aspects of software behavior. For example, control flow, data flow, and program dependency graphs express how the implementation behaves by representing its source code structure. Decision tables and state machines, on the other hand, are used to describe external so-called black box behavior.

Finite state machines are applicable to any model that can be accurately described with a finite number (usually quite small) of specific states. Finite state machines (also known as finite automata) have been around even before the inception of software engineering. There is a stable and mature theory of computing at the center of which are finite state machines and other variations. Using finite state models in the design and testing of computer hardware components has been long established and is considered a standard practice today. Chow [1] was one of the earliest, generally available articles addressing the use of finite state models to design and test software components. Finite state models are an obvious fit with software testing where testers deal with the chore of constructing input sequences to supply as test data; state machines (directed graphs) are ideal models for describing sequences of inputs. This paper introduces model-based testing and discusses its tasks in general terms with finite state models. These FSMs depend on the semantic of a software rather than its structure, it differ from the classical FSM [2], software's input-output specification with trajectory information to describe software behavior. Test generator can easily developed with the proposed model when multi-objective genetic programming techniques are used[2]. This paper is organized as follows. In section 2 an overview is presented of model based testing. Section 3 present an overview for genetic programming and its main problem. In Section 4, the proposed genetic programming is presented. Result and conclusions are given in section 5.

Nada Al Sallami (1971) is with the Department of Management Information Systems, Faculty of economic and business, Al Zaytoonah university of Jordan (e-mail: nada.alsalami@yahoo.com).

II. Model Based Testing (MBT)

Software testing is usually guided by the use of models that represent the system under testing. These models provide information that supports such activities as test case design, test verification and test coverage analysis.

MBT is an efficient and adaptable functional testing technique, supported by the creation and use of a model that describes the behavior of the system under testing. From this behavioral model, test cases can be generated and executed, and the execution results can be evaluated. These models are constructed using software functional requirements and determine the possible actions during a software execution and the expected outputs [3] [4]. MBT is usually divided into the following four activities:

- Elaborate the model that represents the behavior of the software to be tested;
- Select test cases using criteria based on the elaborated model;
- Execute the selected test cases;
- Evaluate the results obtained during the test case execution regarding the expected results.

Depending on the nature of the model used, software testing is called functional testing (also called specification-based testing) or structural testing (also called implementation-based testing). Models used in functional testing are based on software functional requirements while in structural testing are based on the software's internal structure. Nowadays, the most part of software testing is done using functional testing techniques, since they are less costly. However, as research works have been focused on structural testing, software industries do not have many formal ways to perform and assess the test of their products and end up using just heuristics to carry out their software testing [4]. A functional testing technique that has been commonly used in the software industry is Model Based Testing (MBT)[5]. The modeling technique most used to create such models is the Finite State Machines (FSMs). Besides software modeling, the FSMs are used as a basis to define functional testing criteria to be applied in test case selection and adequacy analysis. The FSMs are considered an excellent tool for software modeling, user/developer communication and testing, since their application is simple and intuitive. Although traditional

ISBN: 978-988-19252-1-3
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

FSMs applied in MBT have played an important role in software testing improvement, they do not provide mechanisms to model important behavioral aspects of the software such as its data flow. Due to this limitation, testing techniques based just on traditional FSMs can not make use of further information. For instance, existing functional testing criteria based only on the analysis of FSMs control flow, demand just that elements such as states, arcs and loops are exercised. As a result, the testing effort can achieve poor coverage of software functionality. On the other hand, there are structural testing criteria based on both control flow , and data flow of the software's internal structure, which is represented by program graphs. Therefore, these testing criteria can provide several levels of the internal structure coverage of the software being tested. Such criteria are widely used in the code coverage analysis, during the execution of a test case set [3][5][6].

A. *Finite state machines based testing*

Due to the wide application of FSMs as a modeling technique in MBT, there is a specific testing technique usually called Finite State Machines Based Testing. A model represented by a FSM consists on a state set and a state transition set. Given a current state and an input, the next state and an output can be determined. A sequence of state transitions, from the initial state to any final state, is called Use Scenario. A Use Scenario can be used to describe a general functionality of a modeled system that defines a real use of the system [3][4][5]. The following items are examples of simple functional criteria based on FSMs, used to select test cases in the software industry:

- all-states: it requires that all states inside the FSM must be exercised by at least one test case. Similarly to the structural testing criteria all-nodes, its satisfaction offers a very poor coverage of the FSM;
- all-transitions: it requires that all transitions inside the FSM must be exercised by at least one test case. Similarly to the structural testing criteria all-branches, its satisfaction offers a poor coverage of the FSM;
- all-scenarios: it requires that all possible use scenarios must be exercised by at least one test case. Similarly to the structural testing criteria all-paths, depending on the FSM size, it can be considered impracticable.

III. GENETIC PROGRAMMING

Recently, also more advanced heuristic search techniques have been applied to software testing. These are based on evolutionary algorithms. since their performance in finding test cases was found to be at least as good as random testing, but usually much better. The group of these testing techniques is referred to as evolutionary testing (ET) according to Wegener and Grochtman [7]. ET is an automatic test case generation technique based on the application of evolution strategies, genetic algorithms, genetic programming, or simulated annealing. ET searches for optimal test parameter combinations that satisfy a predefined test criterion. This test criterion is represented through a "cost function" that measures how well each of the automatically generated optimization parameters satisfies the
given test criterion. Evolutionary testing has initially only been applied to traditional procedural software. ET was used to generate input parameter combinations for test cases automatically that achieve, i.e., high coverage, if the test target relates to some code coverage criterion. However, recently, also object oriented software testing based on evolutionary testing has been tackled by researchers [6][7][9].

In genetic programming, genetic algorithm operates on a population of computer programs of varying size and shapes. The space of all possible computer programs is searched for the fittest individual computer program. Computer program structure is defined as the set of all possible composition of functions, which can be composed recursively of the set of functions and terminal. The size and the contents of these computer programs can dynamically change during a run of GP. The function and terminal sets should be selected so as to satisfy the requirements of closure and sufficiency. The programs evolved by GP may be single-part, or multi-part program consisting of a main program and one or more reusable, parameterized, and hierarchically functions called Automatic Defined Function (ADF). ADF can be implemented within the context of GP by establishing a constrained syntactic structure for the individual programs in the population. Two approaches are used to solve complex program :

First: Top-down

- The problem is decomposed into sub-problems
- Solve each of the sub-problems
- Solve the original problem by using the now-available information

The total effort required by this process often proves to be less than the effort required in solving the problem without the aid of the hierarchical process. In addition, if a beneficial decomposition can be found, the solution to the sub-problems will often be reused. Divide and conquer technique is an example of such process.

Second: Bottom-up

- Discover useful regularities and patterns at the lowest level of the problem.
- Change the representation of the problem and restate it in terms of its inherent regularities and patterns to create new problem.
- Solve the presumably more tractable recoded problem.

The bottom-up hierarchical process is considered productive only if the total effort it requires is less than the effort required in solving the problem without the aid of it.

ADF enables GP to solve a variety of problems in a way similar to the three-step hierarchical problem-solving process. GP has two main serious limitations. First , standard program representation so that it is important to choose programming language that is well suited to manipulation by genetic operators. Standard genetic operators like crossover and mutation produce few syntactically correct programs and even, fewer that are semantically correct. It seems that the solution is to devise

ISBN: 978-988-19252-1-3
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

new language-specific genetic operator, that preserves at least the syntactic, and hopefully, the semantic integrity of the programs being manipulated. Second, standard fitness measure so that it is capable of evaluating any computer program that it encounters in any generation of the population. The choice of fitness measurement may be confused [8][10].

IV. THE PROPOSED GP MODEL

The proposed method use FSM model [2][11][12] that describe program's behavior and then use GP to evolve and test such FSM. Our work overcome the above problems by building FSM model that deals with program semantic(to solve presentation problem) and use four new architecture –altering genetic operation(to solve standard fitness measure). The meaning of a program P can be specified by set of function transformation from states to states, as given in[11][12]; hence P effects a transformation on a state vector X, which consists of an association of the variable manipulated by the program and their values. A Program **P** can be defined as 9- tuples, called Semantic Finite State Automata **(SFSA)**: **P=(x, X, T, F, Z, I, O, γ, $X_{initial}$),** where: x is the set of system variables, X is the set of system states, X= { $X_{initial}$, ----, X_{final} }, T is the time scale: T =[0, ∞), F is the set of primitive functions, Z is the state transition function, Z = {(f, X, t): (f, X, t) Є F × X × T , z(f, X, t) = (•X, •t)}, I is the set of inputs, O is the set of outputs, γ is the readout function, and $X_{initial}$ is the initial state of the system: $X_{initial}$ Є X.

Our evolutionary algorithm was defined as 7-tuples: **(IOS, S, F, α_1, T_{max}, β, υ),** as given in [12, 16]**. IOS** is establishing the input-output boundaries of the system. It describes the inputs that the system is designed to handle and the outputs that the system is designed to produce. Syntax Term **(S)** refers to the written form of a program as far as possible independently of its meaning. In particular it concerns the legality or well formedness of a program relative to a set of grammatical rules, and parsing algorithms for the discovery of the grammatical structure of such well-formed programs. S is a set of rules governing the construction of allowed or legal system forms. Primitive Function **(F)**, each function must be coupled with its effect on both the state vector X, and the time scale T of the system. Some primitive functions may serve as primitive building blocks for more complex functions or even sub-systems. Learning Parameter **(α_1)**, is a positive real number specifying the minimum accepted degree of matching between an IOS, and the real observed behavior of the system over the time scale, of IOS. Complexity Parameters **(T_{max}, β)**, T_{max} and β parameters are merits of system complexity: size and time, respectively. It is important to note that there is a fundamental difference between a time scale T and an execution time of a system. T represents system size, it defines points within the overall system, whereas, β, is the time required by the machine to complete system execution, hence it is high sensitive to the machine type. Proof Plan **(υ),** Prove process should be a part of the statement of system induction problem especially when the IOS is imprecise or inadequate to generate an accurate system.

We say P is correct iff it computes a certain function, i.e. P does not loop . Broadly speaking, there have been two main approaches to the problem of developing methods for making programs more reliable: Systematized testing, and Mathematical proof.

A. Systematic testing

Our works use systematized testing approach as a proof plane. The usual method for verifying that a program is correct by testing is by choosing a finite sample of states X_1, X_2,..., X_n and running program P on each of them to verify that: $P(X_1) = f(X_1)$, $P(X_2) = f(X_2)$,……. $P(X_n) = f(X_n)$. are set of tools that support the testing process. First is "Environments", it deals with providing a controlled environment in which testing can take place. Second is "Data Control", the primary issue here is data selection to decide which combination of input values will most thoroughly exercise the system and will most likely uncover defects. Third is "Test Execution", most test tools here work on the final executable code monitoring its operation for conformance to specifications. The specification themselves are analyzed and evaluated for consistency and completeness.

Let I={I_1, I_2, ……., I_j} be an input sequence. The length of I is j. The first element of I is I_1 and the rest of I is the sequence I_2, I_3, , I_j. So that:

First (I) = I_1

Rest(I)=I_2, I_3, …, I_j

Length (I) = j

If I is any sequence of length j, and X is any sequence element, we can make a sequence K of length(j+1) out of X and I, sequence K is denoted by X::I, so that:

First (K) = X

Rest (K)= I

Length (K) = j+1

A sequence K of inputs and an initial state X_{inital} ($X_{initial}$ ::I) give rise to sequence of output as the system run. More precisely, we can define system output as a sequence of outputs generated by I from $X_{initial}$.

Formally, if testing approach is used for system verification, a system proof is denoted υ= (α2, d), where α2 is a positive real parameter defining the maximum accepted error from testing process. α2 focus on the degree of generality, so that α1, and α2, parameters suggest a fundamental tradeoff between training and generality. On the other hand, d represents a set of test cases pairs . Clearly, testing approach is inadequate when the test sample does not exactly coincide with the set of inputs. Although one can never prove that a program is correct by testing, one may perhaps prove that it is incorrect, if $P(X_i) \neq F(X_i)$

B. Mathematical Proof

From a theoretical point of view, one shows that system P is correct by proving a theorem of the form:

$\forall K_i \in N, P(X_i) = f(X_i)$

The proof falls into two parts:

- Proof of partial correctness: A theorem is derived from certain number of axioms by using pre-specified deduction rules. This proof shows that if P terminates, then it gives a correct result.

- Proof or Termination: Proving P terminates for each sequence K_i ∈ P terminates for each sequence K_i is

ISBN: 978-988-19252-1-3
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

necessary to show that each loop of P can only be executed a finite number of times.

Indeed, problems of correctness and termination are closely related. One of the difficulties in program proofs is finding an appropriate invariants for each loop.

C. Architecture Altering Genetic Operations

1) Sub-SFSA Creation

1. An individual is selected based on fitness.
2. Randomly create sub-SFSA defined by a 9-tuples **('x, 'X, 'T, 'F, 'Z, 'I, 'O, 'γ, 'X $_{initial}$),** where:' x is the subset of the corresponding x in the main SFSA, and 'X $_{initial}$ gets its value from the state of the calling transition function.
3. A uniquely named sub-SFSA function 'f is added to the set F of the main SFSA such that each occurrence of 'f in the transition function set z will be replaced by the transition function 'z('f,'$X_{initial}$,1) of the newly created sub-SFSA.
4. Randomly choose a point in the main SFSA transition function and mutate it with 'f.

2) Sub-SFSA Deletion

1. An individual is selected from the population based on fitness
2. Randomly select one sub-SFSA function 'f (if any) from F.
3. Modify invocation of the selected 'f by other functions from F.
4. Delete the corresponding sub-SFSA of 'f.

3) Adding variables to sub-SFSA

1. An individual is selected from the population based on fitness.
2. Randomly select one of the sub-SFSA(if any) from the selected individual.
3. Update 'x term of the selected sub-SFSA by adding new variable to it, such that 'x ϵ x.

4) Deleting variables from sub-SFSA

1. An individual is selected from the population based on fitness
2. Randomly select one of the sub-SFSA(if any) .
3. Update 'x term of the selected sub-SFSA by removing one, randomly selected variable from it.

D. Genetic Program generation Algorithm

1) Initialize the following variables: terms, learning, complexity, generalization, and (δ1, δ2, δ3) parameters.
2) Generate an initial population of random SFSA.
3) Iteratively perform the following operations until the termination criteria is satisfied.
4) Run each individual in the current population and assign fitness value to it using equation 1.
5) Create a new population by applying the following operations on individuals (individual with best fitness value has high probability to be selected):

- Darwinian Reproduction: Simply, the best-of-generation individual is copied into the new population. In the absence of such strategy, it is possible for the best structure to disappear due to sampling error, crossover, or mutation operations.
- Crossover: Three types of points are defined in each individual Z: the state transition function, where :
  - Z = {(f, X, t): (f, X, t) Є F × X × T , z(f, X, t) = (•X, •t)}

  F: the set of primitive functions
  Function arguments: for example the argument of primitive function SUB are A and B written as SUB(A, B) then A or B may be selected as crossover point.
  When crossover is performed, any point type may be chosen as the crossover point of the first parent. The crossover point of the second parent must be chosen only from among points of this type. This restriction to ensures the syntactic validity of the composed offspring
- Mutation: This asexual operation operate on one individual by selecting mutation point type at random, remove whatever is currently at this point and insert randomly generated part. $\mathbf{T_{max}}$ is used to specify maximum length of each newly created offspring[2].

6) 6Apply test plan **(υ),** to the best-of-generation individual, and compute the error **e**.

7) 7 The best-of-generation individual with small error **e**< = **(α_1)**, is designated as the result from the run.

V. CONCLUSION

Because the primary activities of testing, test case identification and design, are typical search problems, they can be tackled by typical search heuristics like GP. Multi objective fitness measure is adopted to incorporate a combination of three objectives: Correctness, Parsimony, and Efficiency. Convergences time is highly sensitive to the initial input-output specification of the program. Multi objective GP can yield a whole set of potential solutions which are all optimal in some sense, and give the engineers the option to assess the trade-offs between different designs. The discussed evolutionary algorithm can be changed to test program in a different implementation language without significantly affecting existing program specification, leading to an increase in the system productivity.

ISBN: 978-988-19252-1-3
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

REFERENCES

[1] Chow T.," Testing design modeled by finite-state machines ", IEE Transaction on Software Engineering, 4(3):178-187, May 1978.

[2] Nada Al Salami, "Analyzing Multi objective Fitness Function with Finite State Automata", "Machine Learning and system Engineering", Lecture Notes in Electrical Engineering, Springer, Vol. 68, 595-606, DOL: 10.1007/978-90-481-9419-3_46, 2010

[3] Marciniak J. J., "Model-Based Software testing", Encyclopedia on Software engineering, Wiley, 2001.

[4] Apfelbaum L., & Schroeder J., " Reducing the Time to Thoroughly Test a GUI ", proceeding of International symposium on software reliability engineering, IEEE Computer Society Press, PP:174-178, 1998.

[5] L. Bottaci and G. Kapfhammer and N. Walkinshaw eds, "Multi Objective Higher Order mutation Testing with Genetic Programming", ", TAIC PART 2009, 4-6 September, p21–29, IEEE press.

[6] Dalal S., Jain A., "Model-Based Testing of a Highly Programmable System " , Proced.

[7] J. Wegener and M. Grochtman, "Verifying Timing Constraints by means of Evolutionary Testing", Real Time System, 3(15), 1998.

[8] Eckart Zitzler, Kalyanmoy, and Lothar Thiele, “Comparison of Multi objective Evolutionary Algorithm: Empirical Result”, Massachustts Institute of Technology, Evolutionary Computation 8 (2) : 173-195, 2000.

[9] J. R. Koza, "Genetic Programming: on the Programming of Computer by means of Natural Selection", Massachusetts Institute of technology, 2004.

[10] Riccardo Poli, William B. Langdon, Nicholas F. McPhee, John R. Koza, “Genetic Programming: An Introductory Tutorial and a Survey of Techniques and applications”, Technical Report CES-475 ISSN: 1744-8050 October 2007. essex.ac.uk/dces/research/publications/.../2007/ces475.pdf

[11] Nada M. A. Al Salami, "**Analyzing Multi objective Fitness Function with Finite State Automata**", "Machine Learning and system Engineering", Lecture Notes in Electrical Engineering, **Springer**, Vol. 68, 595-606, DOL: 10.1007/978-90-481-9419-3_46, 2010.

[12] Nada AlSalami, “**Improving FSM Evolution Algorithm**”, The 2011 International Conference of Computational Intelligence and Intelligent system, World Congress on Engineering 2011, London, U.K., pp. 966-969,

[13] Gross H.,Seesing A., "A Genetic Programming Approach to Automatic Test Generation for Object Oriented Software:, Report TUD-SERG-2006-017.

[14] A.seesing, H. Gross, "A genetic Programming Approach to Automated Test Generation for Object-Oriented Software", TUD-SERG-2006-017, ISSN 1872-5392.

ISBN: 978-988-19252-1-3
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)