Test data generation from UML state machine diagrams using GAs

Test Data Generation from UML State Machine Diagrams using GAs

Chartchai Doungsa-ard*, Keshav Dahal*, Alamgir Hossain*, and Taratip Suwannasart**
*School of Informatics University of Bradford Bradford, United Kingdom
{c.doungsa-ard, k.p.dahal, m.a.hossain1}@bradford.ac.uk
**Department of Computer Engineering Chulalongkorn UniversityBangkok, Thailand
taratip.s@chula.ac.th

Abstract

Automatic test data generation helps testers to validate software against user requirements more easily. Test data can be generated from many sources; for example, experience of testers, source program, or software specification. Selecting a proper test data set is a decision making task. Testers have to decide what test data that they should use, and a heuristic technique is needed to solve this problem automatically. In this paper, we propose a framework for generating test data from software specifications. The selected specification is Unified Modeling Language (UML) state machine diagram. UML state machine diagram describes a system in term of state which can be changed when there is an action occurring in the system. The generated test data is a sequence of these actions. These sequences of action help testers to know how they should test the system. The quality of generated test data is measured by the number of transitions which is fired using the test data. The more transitions test data can fire, the better quality of test data is. The number of coverage transitions is also used as a feedback for a heuristic search for a better test set. Genetic algorithms (GAs) are selected for searching the best test data. Our experimental results show that the proposed GA-based approach can work well for generating test data for some types of UML state machine diagrams.

Keywords- *Test data generation, UML state machine diagram, Genetic algorithm*

1. Introduction

Software testing is a labor intensive and very expensive task. It accounts 50 % of software life cycle [1]. The crucial part of software testing is to select the test data for testing software. The appropriate amount of test data can reduce unnecessary execution time. In addition, the quality of this test data should be good enough; otherwise, test data cannot capture all requirements of the software. Automatic test data generation is proposed to deal with the balance between amount of test data and quality of test data, because random test data generation could not assure the quality of test data.

Automatic test data generation is firstly proposed using control flow graph with a deterministic algorithm to search for a quality test data set [2]. More recently, non-deterministic techniques are also investigated for this problem. Michael, McGraw, and Schatz [3] presented an application of genetic algorithm to solve the same problem reported in [2] to search for a quality test data set from source code with adequate results.

Moreover, having test data before coding can help developers to control their software to conform to the software specification during their development [4]. Software specifications can be UML diagrams, formal language specifications, or any forms of software descriptions.

In this paper, a tool and an approach for generating test data from the software specification using a heuristic technique are proposed. Test data generated by the tool are sequences of actions from the software specification, the UML state machine diagram; the input is a sequence of triggers which can change states of the UML state machine diagram. The quality of test data is measured by the number of transitions which is fired by the input. In essence, this is an optimization problem to find the best sequence of triggers to cover the most transitions. Genetic algorithms (GAs) have been selected as the optimization technique because of its simplicity and effectiveness. In section 2, reviews of automatic test data generation are presented. A short description of the UML state machine diagram and the proposed automatic test data generation framework are given in section 3. The experimental results and discussion are shown in section 4. Finally, section 5 presents the conclusion and future work.

2. Related works

2.1. Software testing

The purpose of software testing is to validate and verify whether the software is working properly following software requirements. The simple technique

is to test software against software requirements. This technique is called black-box testing [5], in which testers prepare test input and expected output which program should return from the given input. The expected input is created from the specification. The test is to compare the value from the expected output and the actual output from program. Another approach is to measure the quality of test data with the source code. The more piece of code test data cover, the more quality it is. This is called white-box testing. The basic white box testing uses coverage criteria as a measurement of test data [6]. Source code is transformed to a control flow graph. The path of the graph which is covered by test data is determined as the coverage criteria. The combination between black-box testing and white-box testing is gray-box testing [7]. Gray box testing measures the quality of test data using the software specification as black-box testing. In addition, gray-box testing also considers the behavior of software specification as white-box testing. In other words, gray box testing has to satisfy the software requirements as black-box testing and use the internal information as white-box testing.

2.2. Test Data Generation from Software Specification

There are many researches on software test data generation on a gray-box testing approach. Clarke [8] proposed an empirical study on automated test data generation using the state based specification EFSMs (Extended Finite State Machines). In his research, the commercial tool TestMaster was used to provide test data from EFSMs specification comparing with providing test data from the software engineer. The result showed that the effort was reduced by 88 % when using the automated test data generation. A state-based specification is not only the specification that is used for automated test data generation. Java Modeling Language (JML) [9, 10], which is a notation to specify the behavior of Java program, is also used for automated test generation. Xu and Yang [11] proposed a framework for generating test data from JML. However, in order to use their technique, developers have to spend some time in studying JML specification.

As Unified Modeling Language (UML) is a main stream in software development [12], generating test data from UML diagrams helps developers to reduce a great amount of their efforts for learning new specifications. Many UML diagrams are selected for automated test data generation. Hong et.al [13] proposed a technique to select the test path from a state diagram. The concept of used and defined variable is applied with the EFSMs which generated from the state machine diagram. The test data generation should be using this generated path but they did not mention about this in their paper. More complete framework for automated test data generation is proposed in [14]. They provided a tool for dividing the UML state diagram into sub scenario. They needed to flatten the diagram and transform it to other forms before generating the test scenario. This approach still needs many transformations before test data can be generated.

To avoid the transformation problem, some researchers proposed heuristic techniques. Cheon, Kim, and Perumandla studied on using the genetic algorithm to generate test data automatically from JML [15]. Genetic algorithms generate and find the best test data for each method which JML specify. They concluded that using genetic algorithm for generating test data is possible. Moreover, their framework is also possible to generate unit testing framework in which developer can use it for JUnit testing.

3. Proposed Test Data Generation Tool

In this section, we discuss related techniques used in our proposed approach. We use a UML state diagram as the software specification and genetic algorithm as the technique for generating test data. The quality of the generated test data is measured by the coverage of transitions occurred by each test data. The design of our approach is discussed at the end of this section.

3.1. UML State Machine Diagram

The UML state machine diagram is used for modeling discrete behavior of an object through finite state transition[16]. A system which is described by the UML state machine contains states in a particular time. Each state in the UML state machine is as same as in other finite state transition system. They are connected by transitions. States of the system can be changed if the system receives a trigger associated with the current states. The trigger will activate the transition which is adjacent with the current states. If the trigger fires transition, the current state will move to the target state.

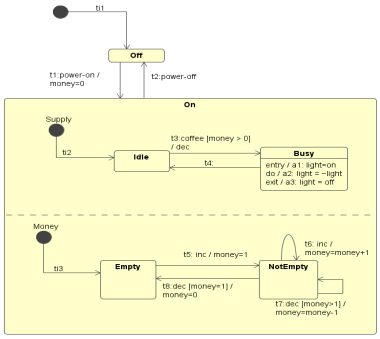

Figure 1. A coffee vending machine state machine diagram

Figure 1 is an example of the UML state machine diagram for a coffee vending machine. When user turns

Table 1. An example of tracing uml state diagram

Sequence: power-on - inc - inc – power-on - inc –dec			
Trigger	State executed	Current state(s)	Attribute status
-	Initial state,ti1, Off	Off	money = 0
power-on	t1, On, Initial state: Supply,ti2, Idle Initial state: Money,ti3, Empty	Idle, Empty	money = 0
inc	t5, NotEmpty	Idle, NotEmpty	money = 1
inc	t6, NotEmpty	Idle, NotEmpty	money = 2
power-on	- (no change)	Idle, NotEmpty	money = 2
inc	t6, NotEmpty	Idle, NotEmpty	money = 3
dec	t7, NotEmpty	Idle, NotEmpty	money = 2

on the machine, the status of machine is changed to "on". If user adds money to the machine, the amount of money is increased. If users request for a cup of coffee, the state machine will check if there is money in the machine, then prepare a cup of coffee. After that the money in state machine is decreased.

A transition is an edge which connects between states. In each transition, there are four components; transition name, trigger, guard condition, and action expression. All of them are optional, which means software designers do not have to specify all of them. For example, in transition "t8:dec[money=1]/money = 0", "t8" is a transition name. "dec" is a trigger. "money = 1" is a guard condition, and "money = 0" is an action expression. A transition name is a name of the transition. It is used for the transition ID. A trigger is a member of a set of events which can occur within the system. For a state machine diagram in Figure 1, members of a set of triggers are "dec", "inc", "power-on", "power-off", and "coffee". The trigger can be fired based on the current state of the state machine diagram and the trigger which is input to the diagram. A guard condition is declared in a block bracket. The guard condition is needed to evaluate to be true if the transition is considered to be fired. Finally, an action expression can be trigger or expression. If an action expression is a trigger, the system acts like there is a trigger from outside the system. If the action is an expression, the expression will be executed.

3.2. Coverage Criteria

The quality of a test data that we evaluate based on the number of coverage the test data can cover. The coverage of test data can be defined by many criteria. Offutt and Abdurazik[17] proposed the coverage level for using with UML diagrams. They proposed four levels of coverage; transition coverage level, full predicate coverage level, transition-pair coverage level, and complete sequence level.

- Transition coverage level is a set of transitions which test data satisfy.
- Full predicate level coverage is a test set that contains test data which examines each clause in each predicate.
- Transition-pair coverage level designs coverage from pairs of adjacent transitions.
- Complete sequence level is a meaningful sequence of transitions to which is designed by the software designer.

In our work, we use transition coverage level for measuring the effectiveness of test data. For example, our generated test data for a coffee vending machine should fire transition "ti1", "ti2", "ti3", "t1", "t2", "t3", "t4", "t5", "t6", "t7" and "t8".

3.3. Tool Design and Implementation

In our proposed approach, the test data to be generated is a sequence of triggers. From the sequence of triggers, the trigger is extracted and used to fire transitions in the state machine diagram. An example of a sequence of triggers and how the trigger fires the transitions in the state diagram depicted in Figure 1 is shown in Table 1.

A sequence of triggers is used as a chromosome of genetic algorithm. In our current approach, the transition coverage level is used for measuring the quality of each test data. In other word, the number of fired transitions is a fitness value of each test data. The test data is a sequence of triggers represented by a GA chromosome. For instance, a sequence in Table 1 covers seven transitions; therefore, the fitness value for the sequence is seven.

Tool to automatically generate the sequence of triggers consists of two parts as shown in Figure 2. The design and implementation of these parts are described below.

3.3.1. UML State Machine Execution

UML state machine execution contains the SDExecutor and DataCollector module. Its roles are to receive state machine information through a chromosome (as described below), and to calculate the fired transitions. The tool evaluates the fired transitions by executing the input triggers. For any given input, the fired transition is evaluated by its attributes, such as a needed trigger, a guard condition. In addition, some transitions contain an action to do when they are fired in an action part of the transition. A constraint of the proposed tool is that the guard condition should be a mathematical expression. Otherwise, the tool cannot

execute it automatically. In order to execute an action declared in transition; the action is in an assignment statement form or a trigger. If an action is a trigger, the tool will use the action as it comes from a sequence of triggers before executing the next trigger in the sequence.

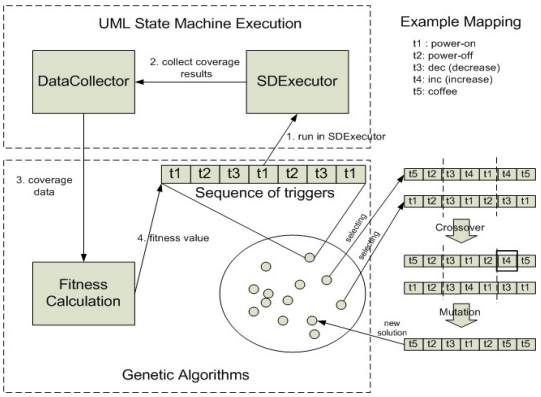

Figure 2. The overview of proposed approach.

The state machine diagram executor (SDExecutor) is responsible for executing the states and the transitions which are changed. The SDExecutor receives a sequence of triggers, and then extracts each trigger to test with the UML state machine diagram. The transition, which is fired by the trigger, is recorded by DataCollector module. DataCollector creates data structure which represents transition path from the given triggers. This data structure is used for calculating fitness value in FitnessCalculation part. FitnessCalculation part calculates fitness value depending on coverage level. Currently, the transition level coverage is used. Finally, the genetic algorithms part is responsible for the genetic algorithm operation.

3.3.2. Genetic Algorithm design

Our proposed approach targets to generate test data set to cover maximum transitions using genetic algorithms technique [18]. GAs are search techniques based on natural genetic and evolution mechanisms which can be used to solve many categories of problems in machine learning and function optimization. GA is iterative procedures which work with a population of candidate solutions (chromosomes). A population of candidate solutions is maintained by the GAs throughout the solution process. Initially a population of candidate solutions is generated randomly or by other means. During each iteration step, a selection operator is used to choose two solutions from the current population. The selection is based upon the measured goodness of the solutions in the population - this is being quantified by a fitness function. The selected solutions are then subjected to crossover. The crossover operator exchanges sections between these two selected solutions with a defined crossover probability. One of the resulting solutions is then chosen for application of the mutation operator, whereby the value at each position in the solution is changed with a defined mutation probability. The algorithm is terminated, when a defined stopping criterion is reached.

Since our approach focuses on finding out a set of triggers to fire most transitions as possible, a chromosome in our genetic algorithm is a sequence of triggers itself. The first population will be generated from all possible triggers in the UML state machine diagram to a sequence of triggers. An example of sequence of triggers for the coffee machine state diagram is shown in the top row of Table 1. The sequence is mapped in to an integer string as show in Figure 2.

The fitness value for each chromosome is calculated from the number of transitions which is fired by the input sequence. If there is any trigger which cannot fire any transition, the tool skips it and gets the next trigger in the sequence. For example, the second “power-on” in Table I. In this case, the tool picks up the next trigger in sequence, which is “inc”

The genetic algorithm part is composed of GAs and FitnessCalculation module. Its functions are: randomly generating the initial population of chromosomes, running the UML state machine execution for retrieving fitness value of each chromosome, implementing crossover operation and mutation operation, and creating new set of chromosomes. The genetic algorithm part is implemented by extending the ECJ module developed by Evolutionary Computation Laboratory, George Mason University [19].

GAs operators used in the tool are the two point crossover and random mutation. Based on some experimentation and previous knowledge on GAs application with other problem [20], the parameters of genetic operation are set as follow:

- The crossover probability is 0.5.
- The mutation probability is 0.05.
- The size of population in each generation is 10.

4. Experimentation

4.1. Case Studies

The following four case studies with some different properties have been done through an experiment set of with the proposed tool:

Case study 1: Coffee vending machine in Figure 1, the explanation of this state machine diagram is in section 3.1.

Case study 2: Enrollment system taken from [21] is shown in Figure 3. An enrollment system diagram describes the activity of the enrollment for each course. The students enroll for the course. When the course is

full, no more students can enroll for the course. The course can be closed for enrollment anytime.

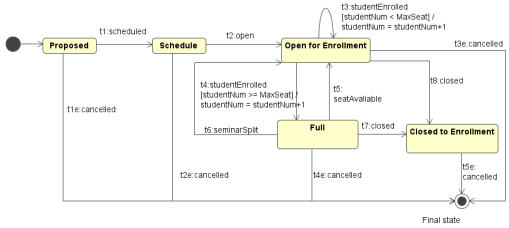

Figure 3. The enrollment state machine diagram

Case study 3: A class management system taken from [21] is shown in Figure 4. The diagram describes a simple course management. Students enroll for the course. While the teacher is teaching, students can drop their course and have the final exam before finishing the course.

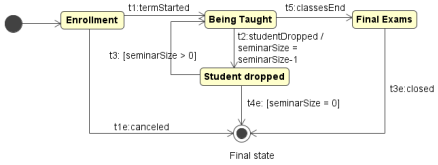

Figure 4. The class management system state machine diagram

Case study 4: A telephone system taken from [16] is shown in figure 5 to demonstrate how the system works. It starts from user lift receiver, waits for the dial tone and starts dialing. It checks every digit for correctness, when every digit is dialed. Then the phone is connected. The user can hang the phone anytime to cancel the operation.

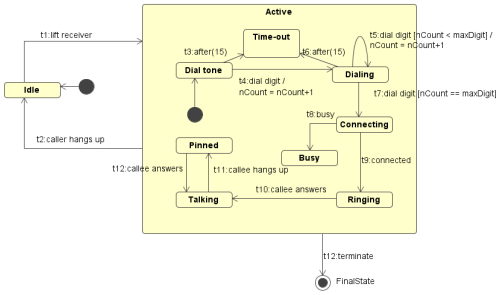

Figure 5. The telephone system state machine diagram

These diagrams have been customized for our tool.

4.2. Result and Discussion

We have experimented all four test cases discussed in the previous section by varying the length of chromosome from 5 to 10. Each time of run, the chromosome which covers maximum number of transitions is the best solution. The GAs approach was run 20 times to eliminate the possibility of being lucky in the stochastic GAs search process. The experimental results are summarized in Table 2. The table shows the average % coverage, and the maximum % coverage given by the best solution of 20 GAs runs for all four case studies using five different chromosome lengths.

From the results, GAs worked well in *case study 1* (coffee vending machine diagram). The best solution is always found when the chromosome length is greater than eight. In *case study 2* (enrollment diagram), *case study 3* (class management diagram) and *case study 4* (telephone system diagram), the average coverage increases, if the chromosome length is increased. However, the best solution for each case is the same for the larger chromosome length (less than 5 in enrollment diagram and top-level class management diagram, and 8 in telephone system).

GAs can obtain 100 % coverage from *case study 1*. The coffee vending machine can work indefinitely because the system does not contain the final state. Without the final state, the coffee vending machine can run through all state transitions; therefore, genetic algorithm can find the optimized sequence of triggers.

The other three case studies contain the final state. Moreover, there is more than one path to reach the final state. Therefore, a single test data cannot cover all transitions. For example, in *case study 3* (class management diagram), one order of fired transitions may be "t1"-"t2"-"t3"-"t5"-"t3e". This sequence will never execute the path "t1e" – i.e. this is an infeasible transition which only one test data cannot reach. In order to overcome the problem of infeasible transitions we need multiple test date set (i.e. multiple sequences of triggers).

Furthermore we need to consider a looping problem while a transition is fired. For example, in *Case study 2* (enrollment state machine), transition "t4" to state "Full" cannot be fired unless the number of students who enroll this course is equal to the maximum seat of the class. The number of enrolled student is increased when transition "t3" is fired. As a consequence, in a sequence of triggers which can cover the transition "t4" must contain triggers which can fired "t3" to make the number of enrolled of students reach the maximum seat number consecutively.

5. Conclusion and future work

In this paper, we propose a framework for using genetic algorithm to generate test data from a UML state machine diagram. The test data to be generated is a sequence of triggers which are fed into the system. The fitness function used in the system is the transition coverage. In our experiment we select only the best solution which covers most of transitions. It is observed that our system works very well with the system which does not contain the final state. The coverage result of

Table 2. An Coverage result for our case studies

Chromosome length	Coffee Vending machine		Enrollment		Top-level Class Management		Telephone System	
	% Coverage		% Coverage		% Coverage		% Coverage	
	Average	*Best*	*Average*	*Best*	*Average*	*Best*	*Average*	*Best*
5	86.36	90.91	32.86	35.71	72.50	75.00	40.67	46.67
6	91.82	100.00	34.29	35.71	75.00	75.00	42.00	46.67
7	99.55	100.00	34.64	35.71	75.00	75.00	45.00	53.33
8	99.55	100.00	35.71	35.71	75.00	75.00	48.00	53.33
9	100.00	100.00	35.71	35.71	75.00	75.00	48.67	60.00
10	100.00	100.00	35.71	35.71	75.00	75.00	52.00	60.00

some examples is not good because of the complexity of the design and the approach we use. Since we select only one best solution, not all transitions can be reached in the diagrams with the final state (i.e. an infeasible transition). In addition, the looping problem is another concern. Some software design requires some transitions to be fired continuously until the attribute reaches some value before transitioning to next state. A subsequence of triggers which can increase or decrease the attribute value to reach the criteria is needed.

Our future work will concentrate on these two problems. The infeasible transition problem may be resolved using a set of test data rather than just one test data. The fitness value should be calculated from the set of test data, not only from the individual test data. We will also like to extend this approach to the other UML diagrams; for example UML class diagram, can be merged with our test data to generate more flexible and usable test data.

6. Acknowledgement

This research has been supported by the EU Asia-link project - TH/Asia Link/004 (91712) -Euro-Asia Collaboration and Networking in Information Engineering System Technology (EAST-WEST).

7. Reference

[1] G. Myers, *The Art of Software Testing, 2 ed*: John Wiley & Son. Inc, 2004.

[2] B. Korel, "Automated software test data generation" *Software Engineering, IEEE Transactions on*, vol. 16, pp. 870-879, 1990.

[3] C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution" *Software Engineering, IEEE Transactions on,* vol. 27, pp. 1085-1110, 2001.

[4] K. Beck, *Test-Driven Development by Example*: Addison-Wesley, 2003.

[5] B. Beizer, *Black-box testing : techniques for functional testing of software and systems*: John Wiley & son Inc., 1995.

[6] R. S. Pressman, *Software Engineering : a practitioner's approach*: McGraw-Hill, Inc., 2000.

[7] N. Q. Hung, T*esting Application on the Web*: John Wiley & Sons, 2003.

[8] J. M. Clark, "Automated Test Generation from a Behavioral Model", *The 11th International Software Quality Week* (QW98), 1998.

[9] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll, "An overview of JML tools and applications" *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS '03)*, ser. Electronic Notes in Theoretical Computer Science, Elsevier, 2003.

[10] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Muller, and J. Kiniry, "*JML Reference Manual*" 2005.

[11] G. Xu and Z. Yang, "JMLAutoTest: A Novel Automated Testing Framework Based on JML and JUnit" in *Formal Approaches to Software Testing, vol. 2931/2004*, Lecture Notes in Computer Science 2004, pp. 70-85.

[12] C. F. J. Lange, M. R. V. Chaudron, and J. Muskens, "In practice: UML software architecture and design description" *Software, IEEE, vol. 23*, pp. 40-46, 2006.

[13] H. S. Hong, Y. G. Kim, S. D. Cha, D.-H. Bae, and H. Ural, "A test sequence selection method for statecharts" *Software Testing, Verification & Reliability, vol. 10*, pp. 203-227, 2000.

[14] L. C. Briand, J. Cui, and Y. Labiche, "Towards automated support for deriving test data from UML statecharts" in *"UML" 2003 - The Unified Modeling Language, vol. 2863/2003,* Lecture Notes in Computer Science: Springer Berlin / Heidelberg, 2003, pp. 249-264.

[15] Y. Cheon, M. Y. Kim, and A. Perumandla, "A Complete Automation of Unit Testing for Java Programs" presented at Proceedings of *the 2005 International Conference on Software Engineering Research and Practice (SERP '05)*, Las Vegas, Nevada, USA,, 2005.

[16] OMG, *"OMG Unified Modeling Language Superstructure version 2.1"* OMG, 2003.

[17] J. Offutt and A. Abdurazik, "Generating Tests from UML Specifications" presented at *2nd International Conference on the UML*, 1999.

[18] T. Bäck, D. Fogel, and Z. Michalewicz, *Evolutionary Computation 1: Basic Algorithms and Operators*: Institute of Physics, London, 2000, 2000.

[19] S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, J. Bassett, R. Hubley, and A. Chircop, "ECJ-A Java-based Evolutionary Computation Research System", vol. 24 February 2006.

[20] K. P. Dahal, C. J. Aldridge, and S. J. Galloway, "Evolutionary hybrid approaches for generation scheduling in power systems" E*uropean Journal of Operational Research, vol. 177,* pp. 2050-2068, 2007.

[21] S. W. Ambler, The Object Primer: *Agile Model-driven Development with UML 2.0: Cambridge* University Press, 2004.