An Automatic Test Data Generation from UML State Diagram using Genetic Algorithm

Chartchai Doungsa-ard, Keshav Dahal, Alamgir Hossain, and Taratip Suwannasart

Abstract— **Software testing is a part of software development process. However, this part is the first one to miss by software developers if there is a limited time to complete the project. Software developers often finish their software construction closed to the delivery time, they usually don't have enough time to create effective test cases for testing their programs. Creating test cases manually is a huge work for software developers in the rush hours. A tool which automatically generates test cases and test data can help the software developers to create test cases from software designs/models in early stage of the software development (before coding). Heuristic techniques can be applied for creating quality test data. In this paper, a GA-based test data generation technique has been proposed to generate test data from UML state diagram, so that test data can be generated before coding. The paper details the GA implementation to generate sequences of triggers for UML state diagram as test cases. The proposed algorithm has been demonstrated manually for an example of a vending machine.**

Index Terms— **test data generation, gray-box testing, artificial intelligence, genetic algorithm**

I. INTRODUCTION

Software testing is an important activity to assure the quality of software. Unfortunately, software testing is very labor intensive and very expensive. It can take about 50 percents of total cost in software developing process [1]. Automated test data generation reduces an effort of software developers for creating test cases. The software testers may need to spend a longer time using many test cases if the test data used are not of high quality. Therefore, a performance of executing test data is an important issue to reduce the testing time.

Software testing is usually the first part of software development stages, which software developers decide to omit when there is a limited time to deliver the software. In other word, developers may not have enough time after they finished their coding to create test cases to test their code. Generating test cases before coding can resolve these problems. This not only helps developers to test their program when they finish coding but also controls the developers to program the software as defined in the software specification [2]. In this case the software specifications are the main sources for generating test cases as these documents describe the software system to be developed in detail. Software specifications may be UML diagrams, formal language specification, or natural language description.

In this paper, an approach for generating test data from software specification using heuristic technique is proposed. Our approach uses heuristic technique for determining appropriate test data for testing software. Genetic algorithm is selected due to its effectiveness and simplicity. Test data is generated from UML state diagram. In section 2, a review of testing problem and some automated test data generation techniques are presented. A short description of UML state diagram and an automatic test data generation technique are given in section 3. Section 4 illustrates an example of the use of the proposed approach. Finally, section 5 presents conclusion and future work.

C. Doungsa-ard is with Bradford University, Bradford, West Yorkshire, BD7 1DP, UK, on leave from College of Art, Media, and Technology, Chiangmai University, Chiangmai 50200 Thailand. (phone: 665-389-3216; fax: 665-389-3217; e-mail: chartchai.d@ camt.info).

K. Dahal is with Bradford University, Bradford, West Yorkshire, BD7 1DP, UK (e-mail:K.P.Dahal@bradford.ac.uk)

A. Hossain is with Bradford University, Bradford, West Yorkshire, BD7 1DP, UK (e-mail:M.A.Hossain1@bradford.ac.uk)

T. Suwannasart., is with deparment of computer engineering, faculty of engineering, Chulalongkorn university, Bangkok 10330, Thailand (e-mail: Taratip.s@chula.ac.th).

This work is supported by EAST-WEST project (Euro-ASia CollaboraTions and NetWorking in Information Engineering System Technology) Ref: TH/Asia Link/004 (91712)

II. RELATED WORKS

A. Software Testing

During a testing stage, software testers use test data as input to drive the software program. The testers usually have an expected outcome from these data, which is also test oracle. It may include a return value, a sequence of method calls, or any types of output that the testers want to inspect. After program has been executed, it returns output which is an actual output. Testers have to compare between test oracle and actual result to decide whether a program executes correctly for the given test, and make a verdict of "pass" or "fail".

If testers want to test functional requirements, they may use black-box testing technique. Black-box testing [3]does not need knowledge of how software is programmed. Test oracles are specified by software design or software specifications. Testers inject test data to execute program, then compare actual result with the specified test oracle. By contrast, white-box testing needs knowledge of how software is programmed. In white-box testing, paths or statements which has been executed are test oracle. These are called coverage criteria. There are three main types of coverage criteria: statement coverage, branch coverage, and path coverage. Statement coverage reports whether each statement is encountered by the test suite or not. Branch coverage reports whether every

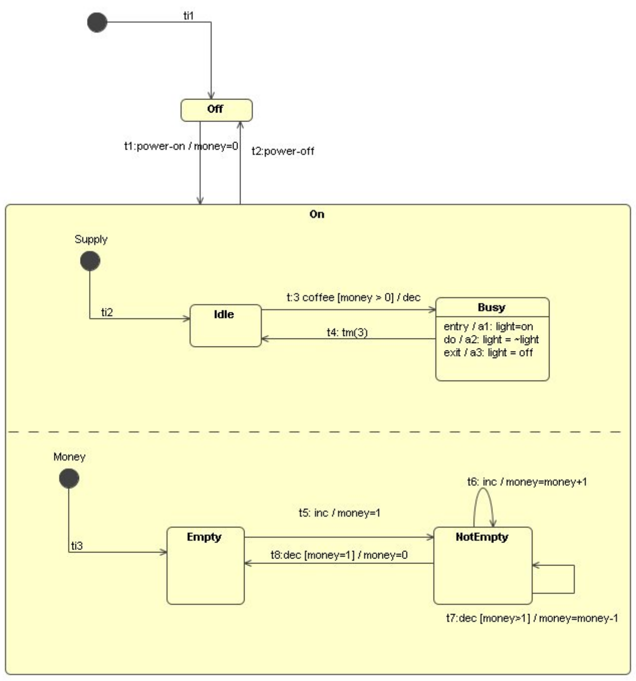

Fig. 1. An example of vending machine state diagram

branch structure (if – else clause or while clause) has been executed for true and false condition in each branch. Finally, path coverage reports whether all possible paths in function has been tested.

In Object-oriented context, the structure of software is more complicated than the structural one. Conventional test approaches may not be enough for testing. The combination of those two traditional approaches is called Gray-box testing [4]. In Gray-box testing, test data generates based on the high level design which specifies the expected structure and behavior of system. Gray-box testing investigates the coverage criteria of white-box method and finds all possible coverage paths. In addition, the generated test case should be satisfied with functional requirement as in the black-box testing criteria.

B. Test Data Generation on Gray-Box Techniques

Many automated test data generation techniques produce test data based on gray-box method. Not only does gray-box testing concern functional requirement as black box testing, but also concerns on behaviors of system. Clarke [5] proposed an empirical study which compared efforts between automate test generation and manual test generation. In his report test data was generated from extended finite state model (EFSM). The research shows that the automate test data generation could reduce an effort from manual test data generation for more than 88 percents. Xu and Yang [6] proposed test data generation framework called JMLAutoTest framework. JMLAutoTest framework generates test data from Java Modeling Language(JML) [7, 8]. JML is a notation for specifying behavior and interface in Java class and method. Since JML is a formal specification, developers should spend efforts to understand JML before writing specification.

Because UML diagrams are now widely used for software development [9], generating test data from UML diagrams should help developer to reduce a great number of effort. Wang, et.al [10] proposed test data generation from activity diagram. They extracted a test scenario from activity diagram. The test scenario is a sequence of possible paths in activity diagram. From these paths, the executing sequence of program has been generated in order to cover all possible paths. However, activity diagram describes flows of system, not the behavior of the system.

Due to performance of generating test data and a concern of size of test data set, heuristic techniques are applied for test data generation. GADGET [11] and TGEN [12] use genetic algorithm to improve quality of generating test data. GADGET generates test data from a control flow graph generated from source code. A fitness function is defined for each condition node in control flow graph. An empirical study showed that test data generated by GADGET covers more than 93 percents of source code, while random testing achieves around 55 percents.

TGEN transforms a control flow graph to a control dependency graph (CDG). Each part of CDG represents the smallest set of predicate to traverse every node in control flow graph. Both GADGET and TGEN generate test data using white box method; therefore, test data can be generated only after software is finished.

Using Genetic algorithm to generate test data from software model is proposed in [13]. JML is a model for generating test data. Fitness function is calculated by coverage of paths and post condition defined by JML.

III. A TEST DATA GENERATION FROM UML STATE DIAGRAM USING GENETIC ALGORITHM

A. UML State Diagram

In our approach, we focus on generate test data from UML state diagram [14]. UML state diagram is a graph-like diagram. It describes the system in a state machine. The system has states at a time. States of system are changed due to an event trigger that happens to the system. Trigger and attributes of system specify the transition of current states into new states. Fig 1 is an example of state diagram of a coffee vending machine.

The coffee vending machine starts from “off” state. When user turns on the machine, the status of machine is changed to “on”. If user adds money to the machine, then a money sub state is moved from “Empty” state to “notEmpty” state and attribute “money” is increased. At this moment, money is greater then zero; therefore, if a user requests for coffee, the state of the vending machine in supply part is moved from “Idle” to “busy” to give the user a glass of coffee and attribute “money” is reduced. After finishing coffee preparation, the status of the machine is returned to “Idle”.

From the example, every state contains a state name and transitions. The state name is used for specifying a particular state. Transition defines how status of the system is changed. From the transition “t3:coffee[money > 0]/dec”, t3 is a name of transition. Coffee is an event trigger name, this mean that if the system is “On” an “Idle” state and user makes a coffee event-trigger, state may change form ‘Idle” to “Busy”, but

TABLE 1
AN EXAMPLE OF TRACING UML STAE DIAGRAM

Sequence: power-on - inc - inc - tm - inc - tm

Trigger	State executed	current state	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, noEmpty	Idle, NoEmpty	money = 1
inc	t6, noEmpty	Idle,NoEmpty	money = 2
tm	end		
inc			
tm			

guard condition should be considered before state changed. Some event triggers may consist of parameters. These parameters are for changing attributes of system. Guard condition is an option for each transition. If it is specified, it is defined in block bracket. Guard condition may be mathematical expressions or sentences that can be evaluated as be true or false. If sentences or statements in guard condition are true, the state of the system is changed to a target state of this transition. The last component of transition is an action. An action is an executable and atomic computation. It may be a statement which changes attribute of the system, an event trigger for the system itself, or other system. In the example, an action for transition "t3" is placed after "/" symbol which is "dec". "dec" is a event trigger for this system. "dec" may trig a transition "t7".

B. Genetic Algorithm

Our proposed approach targets to generated test data set which covers maximum states and transitions using a genetic algorithm (GA) technique [15]. GA is a search technique 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 an iterative procedure which works with a population of candidate solutions (chromosomes). A population of candidate solutions is maintained by the GA 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. A simple one-point crossover operation is shown in Fig 2. 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. An example of using mutation operator is shown in Fig 3. The algorithm is terminated, when a defined stopping criterion is reached.

C. GA implemetation

A number of decisions must be made in order to implement the GA for test data generation. There are problem specific decisions which are concerned with the search space (and thus the chromosome representation) of feasible solutions and the form of the fitness function.

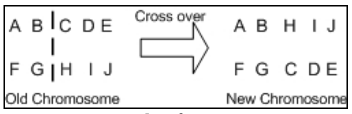

Fig. 2. An example of crossover operation.

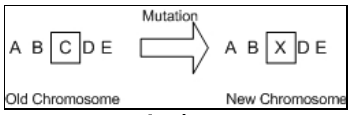

Fig. 3. An example of mutation operation.

We propose to use a sequence of triggers for UML state diagram as a chromosome. The sequence of triggers is an input for the state diagram. Our approach search for a state and transition coverage occurred by the sequence. The sequence of triggers is extracted one by one for tracing for the coverage. When the first trigger is consider, the trigger is determined whether it can activate any transitions connected to the current status. If the trigger can make a transition from current state, the current state is moved to new state. Then next event trigger is selected to consider again and so on. If no transition can be made, tracing for the state coverage will be stopped and the state and transition coverage are recorded without taking the rest of the sequence to consider. Extracting new event trigger in a sequence after a trigger which cannot make any transition in state may break an order of reaching state. For example, in case of Fig. 1 if a chromosome is shown as a sequence of "power-on, inc, tm, dec", after "inc" trigs, current state is "Idle", and "notEmpty". From this status "tm" can not make any new coverage state or transitions. If "dec" is considered to unveil new coverage transition which is "t7", it is look like "power-on, inc, tm, dec" can cover transition "t7". But "tm" trigger does not involve any transitions which help to reach transition "t7". Omitting that trigger may make the length of chromosome changed, suggesting to use a GA with variable chromosome length.

The coverage of state and transition is considered as shown in Table 1. A sequence of trigger is power-on - inc - inc - tm - inc – tm, which is generated randomly. The sequence is applied for revealing state and transition coverage as follows. Starting from initial state, there is one transition connected to initial state and there is no event trigger on that transition. Hence, current state has been move to off. Then, the tool extracts the first trigger in the sequence, "power-on" to execute. With event trigger "power-on", there is a transition from state "off", which needs "power-on" trigger. That is a

```
Reachable transition is a transition which its source state can be executed by current test cases.
Reachable transition source is a state which is a source of reachable transition
Test cases: set of sequences of trigger to run from the beginning of state diagram.
1. Generate test cases randomly, keep it in repository
2. Run test case in state diagram
3. Collect all coverage then store in Coverage table, keep history of attribute changed in state diagram
4. If all transitions and states are covered go to End.
5. Select one transition which are reachable, and have not been marked as covered; by
   5.1 if there are transitions which do not have a guard condition, select one of them,
       Else select transition randomly.
6. Defined objective function as follow: objective function for test case is aW + bX+ cY + Z
   Where
                  a, b,c are constant value where a = 0 when there is no guard condition in selected transition
                  W is a number of states in test cases which value of attribute in that state make guard
                     condition to be true.
                  X is a number of transitions which is covered by this test but have not been covered by previous test
                     set
                  Y is a number of states that can be reached by test case to reachable transition source.
                  Z is a number of state and path coverage for the test case.
7. Select initial test data by ranking objective function from the repository,
8. If the initial population is not enough, randomly generate them
9. Repeat GA using above objective function state in 6 until all uncovered transition are covered, or there are no new
         covered transition for some times
   9.1 Store all test cases generated in repository.
10. Mark selected transition as subjected to GA
11. If there is some transition which is not covered by previous test cases and have not marked as subjected to GA and
         reachable. goto 5
    Else End system.
```

Fig. 4. An algorithm for proposed approach

TABLE 2

AN EMPIRICAL STUDY OF PROPOSED APPROACH

iteration	covered states and transtions	number of coverage states	number of coverage transition
Initial	Initial state, off, on, Idle, InitMoney, InitSupply, Empty ti1, ti2, ti3, t1	7	4
1	Initial state, off, on, Idle, InitMoney, InitSupply, Empty, notEmpty ti1, ti2, ti3, t1, t5, t6, t7, t8	8	8
2	Initial state, off, on, Idle, InitMoney, InitSupply, Empty, notEmpty ti1, ti2, ti3, t1, t5, t6, t7, t8	8	8
Since round 1,2 does not covered any new states, or transition. But there are some states and transitions which were not covered; therefore, list of state and transition is stored in a coverage repository.			
3	Initial state, off, on, Idle, InitMoney, InitSupply, Empty, notEmpty ti1, ti2, ti3, t1, t2, t5, t6, t7, t8	8	9
4	Initial state, off, on, Idle, InitMoney, InitSupply, Empty, notEmpty,Busy ti1, ti2, ti3, t1, t2, t3, t4 ,t5, t6, t7, t8	9	11

transition "t1". Current state is now moved to state "on". Now there are two sub initial states in state "on", therefore, current state has been moved to two initial states. These two initial states bring system to new current states "idle" and "empty". Next event trigger has been taken, "inc". Transitions from state "idle" and "empty" is checked whether they contains a trigger "inc". From the diagram, transition "t5" from "empty" state contains "inc" trigger, as a consequence, current state is moved from "empty" to "notEmpty". Moreover, during the transition of "t5", there is an action declared in diagram as "money = 1", so an attribute of system has been set as money = 1. "Idle" state is still a current state because "inc" does not make any changes to "idle". So, current statuses are "noEmpty" and "Idle". Next event trigger is "inc". "Inc" drives transition "t6" and changes current status. Transition "t6" is a loop transition, so, the current status from "noEmpty" is still "noEmpty". Next event trigger is "tm". Unfortunately, event "tm" does not drive any transitions, which are connected state "noEmpty" and "idle". The search for coverage is stopped, because no new state is covered.

Fitness function is calculated by a number of states and transitions covered by the chromosome. In addition, states and transitions, which are covered by a pool of chromosome in each generation, are recorded. If a chromosome in a new generation covers new state or transition has not been covered before, there will be an addition score for that chromosome. These values used for analysis in selecting function and evolutionary operation. An algorithm of proposed system is shown in Fig 4.

IV. EXAMPLES

In this section we demonstrate the application of the proposed algorithm manually to generate test data for the state diagram shown in Fig 1. In this experiment, we fix the size of pool to 6. While the crossover probability used is 0.4, the mutation probability is around 0.3. The mutation probability is set high in order to evaluate the technique manually.

Firstly, a list of coverage states and transitions should be created as a coverage repository. For the initial round, the coverage repository is empty. Then chromosomes are generated randomly, and stored in an initial pool. After all sequences are executed, a list of coverage states and transitions is recorded. Fitness value is calculated for each chromosome using the fitness function. Then a selction operator based on the fitness vales is used to select chromosomes for crossover and mutation operators. After a new generation of chromosome is generated, UML state diagram is executed again.

In the third round, the generated sequence could not cover any new states or transitions, then a list of coverage states and transition is added to the coverage repository. A new fitness function is generated. The new function gives an extra score for a sequence which covers states or transitions which are not stored in the coverage repository. Then, a new generation of chromosomes is generated.

In this experiment, GAs ran for two generations to cover all states and transitions. Table 2 shows detail of each iteration.

V. CONCLUSION AND FUTURE WORK

An approach for generating test data from UML state diagram using genetic algorithm is proposed. This approach will help software developers to reduce their effort in generating test data before coding. In order to create an effective and robust solution we have demonstrated how genetic algorithm can be applied as a concept.

A proof-of-concept tool development is in progress. The tool contains two parts. The first part is for executing or tracing for coverage states or transitions when an input has been given to an UML state diagram. A search for coverage part is described in section III. The second part of the tool is for calculating fitness values and GA evolution process. We expect this approach to be a useful tool for software developers to help generating test data from UML state diagram to test their software program.

VI. 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).

VII. REFERENCES

[1] Myers, G., *The Art of Software Testing*. 2 ed. 2004: John Wiley & Son. Inc. 234 pages.

[2] Beck, K., *Test-Driven Development by Example*. 2003: Addison-Wesley. 220.

[3] Beizer, B., *Black-box testing : techniques for functional testing of software and systems*. 1995: John Wiley & son Inc. 294.

[4] Hung, N.Q., *Testing Application on the Web*. 2003: John Wiley & Sons.

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

[6] Xu, G. and Z. Yang. *JMLAutoTest: A Novel Automated Testing Framework Based on JML and JUnit*. in *Lecture Notes in Computer Science*. 2004.

[7] Burdy, L., et al. *An overview of JML tools and applications*. in *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS '03), ser. Electronic Notes in Theoretical Computer Science*. 2003. Elsevier.

[8] Leavens, G.T., et al., *JML Reference Manual*. 2005.

[9] Lange, C.F.J., M.R.V. Chaudron, and J. Muskens, *In practice: UML software architecture and design description.* Software, IEEE, 2006. **23**(2): p. 40-46.

[10] Wang, L., et al. *Generating test cases from UML activity diagram based on Gray-box method.* in *Software Engineering Conference, 2004. 11th Asia-Pacific* 2004.

[11] Michael, C., G. McGraw, and M.A. Schatz, *Generating software test data by evolution.* Software Engineering, IEEE Transactions on, 2001. **27**(12): p. 1085-1110.

[12] Pargas, R., M. Harrold, and R. Peck, *Test-data generation using genetic algorithms.* Software Testing, Verification and Reliability, 1999. **9**(4): p. 263-282.

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

[14] OMG, *OMG Unified Modeling Language Specification version 1.4.2*. 2001: OMG.

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