ADOPTING GENETIC ALGORITHM TO ENHANCE STATE-SENSITIVITY PARTITIONING

Ammar Mohammed Sultan[1], Salmi Baharom[2], Abdul Azim Abd Ghani[3], Jamilah Din[4], and Hazura Zulzalil[5]

[1]*Universiti Putra Malaysia, Malaysia, ammar.alsultan@hotmail.com*
[2]*Universiti Putra Malaysia, Malaysia, salmi@upm.edu.my*
[3]*Universiti Putra Malaysia, Malaysia, azim@upm.edu.my*
[4]*Universiti Putra Malaysia, Malaysia, jamilahd@upm.edu.my*
[5]*Universiti Putra Malaysia, Malaysia, hazura@upm.edu.my*

ABSTRACT. Software testing requires executing software under test with the intention of finding defects as much as possible. Test case generation remains the most dominant research in software testing. The technique used in generating test cases may lead to effective and efficient software testing process. Many techniques have been proposed to generate test cases. One of them is State Sensitivity Partitioning (SSP) technique. The objective of SSP is to avoid exhaustive testing of the entire data states of a module. In SSP, test cases are represented in the form of sequence of events. Even recognizing the finite limits on the size of the queue, there is an infinite set of these sequences and with no upper bound on the length of such a sequence. Thus, a lengthy test sequence might consist of redundant data states. The existence of the redundant data state will increase the size of test suite and consequently the process of testing will be ineffective. Therefore, there is a need to optimize those test cases generated by the SSP in enhancing its effectiveness in detecting faults. Genetic algorithm (GA) has been identified as the most common potential technique among several optimization techniques. Thus, GA is investigated for the integrating with the existing SSP. This paper addresses the issue on how to represent the states produced by SSP sequences of events in order to be accepted by GA. System ID were used for representing the combination of states variables uniquely and generate the GA initial population.

Keywords: genetic algorithm (GA), state-sensitivity partitioning (SSP), test case, sequence of events, data state

INTRODUCTION

Software testing is the most costly and time consuming phase in software development lifecycle. It consumes about 50% of the software development cost (Pressman, 2010). In general, research in software testing can be categorized into three categories which are test case generation, test execution and test oracle. However, test case generation was found to be the dominant among those three categories. Many techniques have been proposed for generating test cases in order to improve the effective and efficiency of detecting faults. One of them is State Sensitivity Partitioning which is also known as SSP technique (Baharom & Shukur, 2008; Baharom & Shukur, 2010; Baharom & Shukur, 2011). It was built based on Parnas formal specifications to test a module that consists of one or more access programs which

share the same data structure. The output of a module is based on the event triggered, the value of input parameters, conditions and actions. Thus, test data for a module might consist of event sequences (or test sequences) rather than single events. In order to avoid exhaustively testing the entire data states of a module, SSP partitions the entire data states based on the state's sensitivity towards events, conditions and actions.

There are six sequential steps in SSP technique which are: (i) identifying sensitive access program, (ii) partitioning states into equivalence classes, (iii) constructing a state transition model, (iv) selecting test cases based on all-transition coverage criteria, (v) adding insensitive event at the end of each selected test cases and (vi) applying boundary value analysis (BVA) technique to the selection of input parameters. Each test case selected in step four (4) must be represented by at least one sequence of events. In SSP, the sequence of events is selected randomly and thus, any sequence of events is valid as long as it follows the specified conditions of the constructed state transition model in step three (3). For example, a sequence of events for a queue test case when trying to add item into a full queue might include adding twenty items onto the queue; removing eighteen items, adding fifty more, removing fifty two, adding ten more, removing ten, adding one more and checking the result. Hence, the sequence of events can be very lengthy and might contain redundant data states. The lengthy sequence with redundant states obviously makes testing expensive and relatively ineffective.

In the literature, many optimization techniques have been suggested. One of the technique is search techniques (Alsmadi, Alkhateeb, Maghayreh, Samarah, & Doush, 2010; Kulkarni, Naveen, Singh, & Srivastava, 2011) and genetic algorithm (GA) has been identified as the most common search technique employed for generating test cases (Ali, Briand, Hemmati, & Panesar-Walawege, 2010). The success stories of GA inspired us to adopt GA in our work. The adoption of GA requires the state produced by SSP sequence of events to be represented in a form that can be accepted by the GA. Thus, this paper describes the on-going research that addresses the issue on how to represent the states. The remainder of this paper is organized as follows: an overview of SSP is presented in the next section; followed by a general overview of the GA search technique and its application. Next, the states representation is being outlined. Finally, the last section summarizes the paper along with the conclusion.

STATE SENSITIVITY PARTITIONING (SSP)

A module may consist of one or more access programs that share a data structure and its behaviour is depending on the event triggered, the value of input parameters and conditions. Generating test cases from such information might involve large number of data states, as the number of states grows exponentially in the number of program variables. For example, as described in (Gannon, Purtilo, & Zelkowitz, 1994), in order to test correctness of two variables A and B of 32 bit integers, one needs to perform 2^{32} X 2^{32} which is approximately 10^{20} tests. Hence, it would require more than 30,000 years of testing with the assumption 10^{8} tests per second. Therefore, it is impossible to explore the entire state space with limited resources of time and memory.

State Sensitivity Partitioning (SSP) is a test case generation technique that was introduced by Salmi and Shukur (Baharom & Shukur, 2008; Baharom & Shukur, 2010; Baharom & Shukur, 2011). SSP was proposed to generate test cases of a module. In order to avoid testing the entire data states of a module, the states are partitioned based on state's sensitivity towards events, conditions (i.e. pre-conditions) and actions (i.e. post-conditions). The goal is to partition those data states in such a way that each data state in a partition behaves similarly towards access-programs (events), conditions and actions (either sensitive or insensitive). There are six sequential steps in performing the SSP technique that are identifying sensitive access program, partitioning the states into equivalence classes , constructing a state transition model,

selecting test case based on all-transition coverage criteria, adding insensitive event at the end of each selected test cases and applying boundary value analysis (BVA) technique to the input parameters.

We illustrate the idea of SSP technique using a circular queue example. A circular queue consists of three access programs which are: add(), remove() and front(). In SSP, both add() and remove() are identified as sensitive access programs as they modify the data states during their execution. In contrast, front() is identified as insensitive access program as it does not modify the data state. The entire data states are partitioned into four possible equivalence classes based on the number of identified sensitive access programs. In the third step, a state transition model is constructed as presented in Figure 1.

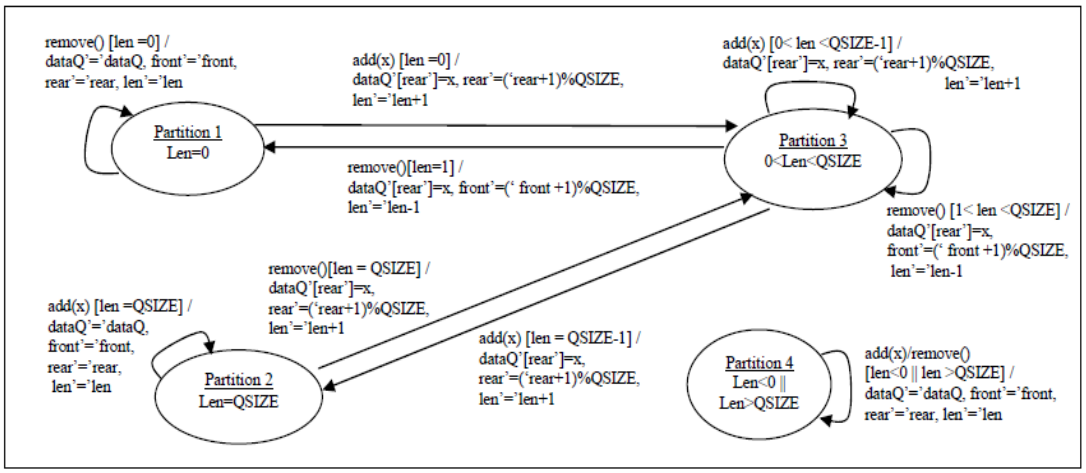

Figure 1. State transition model of Circular Queue

Once the state transition diagram is constructed, test cases are selected based on all-transitions coverage criteria where each transition represents one test case. Table 1 lists the ten test cases obtained from the state transition model. Each of the test cases then must be represented by at least one test data that is in the form of test sequence.

Table 1.The test cases of Circular Queue program

#	P	Event	Pre-Condition	Post-Condition
1.	1	**Add**	len = 0	dataQ'[rear']=x, rear'=('rear+1)%QSIZE, len'='len+1
2.	1	**Remove**	len = 0	dataQ'='dataQ, front'='front, rear'='rear, len' = 'len
3.	2	**Add**	len = QSIZE	dataQ'='dataQ, front'='front, rear'='rear, len' = 'len
4.	2	**Remove**	len = QSIZE	dataQ'[rear']=x, front'=('front+1)%QSIZE, len'='len–1
5.	3	**Add**	0<len<QSIZE – 1	dataQ'[rear']=x, rear'=('rear+1)%QSIZE, len'='len+1
6.	3	**Add**	len = QSIZE – 1	dataQ'[rear']=x, rear'=('rear+1)%QSIZE, len'='len+1
7.	3	**remove**	1<len<QSIZE	dataQ'[rear']=x, front'=('front+1)%QSIZE, len'='len–1
8.	3	**remove**	len = 1	dataQ'[rear']=x, front'=('front+1)%QSIZE, len'='len–1
9.	4	**Add**	len<0 && len>QSIZE	dataQ'='dataQ, front'='front, rear'='rear, len' = 'len
10.	4	**remove**	len<0 && len>QSIZE	dataQ'='dataQ, front'='front, rear'='rear, len' = 'len

Referring to selected test cases of SSP in Table 1, below are some examples of sequences for the test cases with assumption of QSIZE=5. Following the fifth step in SSP, insensitive event (front()) is added at the end of each test cases. Lastly, in the sixth step, the BVA technique is applied as the value of input parameter.

TC1: `_.add(1).front()`
TC2: `_.remove().front()`
TC3: `_.add(1).add(-1).remove().add(1295644148).add(-1295644148).front()`
TC4: `_.add(0).add(Integer.Max_value).add(Integer.Min_value).add(1).add(-1).front()`
TC5: `_.add(0).add(1).remove().front()`
TC6: `_.remove().add(0).front()`

In SSP, the sequence of events is selected randomly and thus, any sequence of events is valid as long as it follows the specified conditions of the constructed state transition model. For example, a sequence of events for a queue test case when trying to add item into a full queue might include adding twenty items onto the queue; removing eighteen items, adding fifty more, removing fifty two, adding ten more, removing ten, adding one more and checking the result. Hence, the sequence of events can be very lengthy and might contain redundant data states. The lengthy sequence with redundant states obviously makes testing expensive and relatively ineffective. Also, there is redundancy that occurs between two or more test sequences (i.e. sequence of events) where a test sequence may appear as subset of other test sequence. Therefore, finding appropriate technique to optimize the test suite by removing redundant data states will be our main focus. Among the available techniques that can be used for this purpose, search techniques are the most common for obtaining optimized test suites.

GENETIC ALGORITHM (GA) APPLICATION

The importance of software testing attracts more application of search techniques with the goal of saving effort and time. Among all search techniques for test cases generation, GA is the most common. GA is a population based metaheuristic technique that follows the theory of natural evolution by Darwin. In GA, the optimal solutions evolved through applying reproduction and selection operations on populations over successive generations (John, 1975). The typical GA consists of five repetitive steps that last till finding an optimum solution or reaching the maximum number of iterations, which called termination criteria. The steps are: 1) random initialization of population that contains candidate solutions. Each solution is represented as a chromosome or sequence of variables/parameters (Li, Harman, & Hierons, 2007); 2) evaluation of new candidate solutions, if the termination criterion is not met; 3) selection of promising candidate solutions based on fitness function. Fitness function is used for evaluating whether the solution is able to solve optimization problems or not; 4) crossover application, two chromosomes are taken for generating offspring through recombination (Mühlenbein, hlenbein, & Schlierkamp-Voosen, 1993); 5) mutation through operators for altering one gene or more.

The parallelism nature of search in GA leads to fast calculations and, hence, makes it effective for solving non-linear, multi-modal and discontinuous problems. Consequently, software testing dominates GA applications compared to other SDLC phases. This includes different disciplines such as test cases generation (Ali, Briand, Hemmati, & Panesar-Walawege, 2010; McMinn, 2004), test cases prioritization within test suites (Conrad, Roos, & Kapfhammer, 2010), and test suites reductions (Li et al., 2007).

However, applying GA to optimize the test cases that are composed of sequences of events requires an extensive care due to the special nature of data. Besides, the invocation of each event in the sequence may lead to different states. Therefore, there is a need to grasp the changes of states and represent them in a way that GA can handle. In next section, the representation of states is described.

THE STATES REPRESENTATION

For the sake of representation, the problem has to be represented in a way that facilitates searching for solutions. Even though binary representation is the most preferable, other representation forms can be used, such as: gray, real numbers, graphs and trees.

In terms of representation, the data features SUT states. A state is composed of all the variables that are going to be modified by sensitive events in SSP. For example, a state for the Circular Queue case study is composed of four variables: *len*, *rear*, *front* and *dataQ*. The former variable is of integer type for indicating the number of items in the array; *rear* and *front* are two integer variables that pinpoint to the tail and head, respectively; while the latter is an integer array that is supposed to store the items added to the queue.

Regarding the values that can be used, *len* ranges from 0 to max queue size. On the other hand, *rear* and *front* can have values range from 0 to max-1. Besides, the maximum number of items to be stored are equal to the max queue size whilst the potential values are based on the BVA technique. So, a state for circular queue has a form of: (*len*; *rear*; *front*; *dataQ*). For instance, the `_.add(0).add(1).remove().front()` test case produces three different states in addition to the initial state, as in Figure 2 (assuming the max queue size=10).

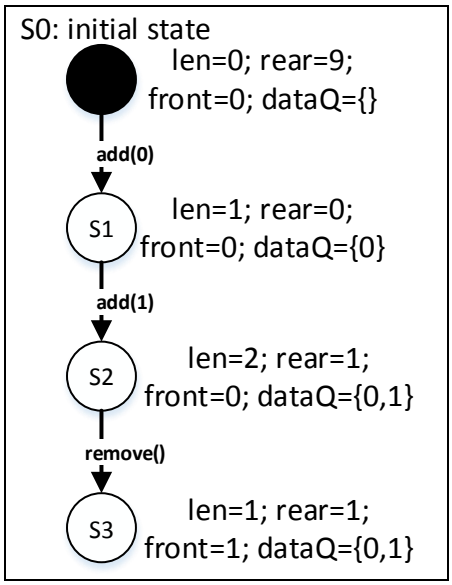

Figure 2. States flow

Apparently, the sequences of events dictate the states to be generated. In other words, different sequences lead to different states even if they contain the same events. For example, `_.add(0).add(1).front()` and `_.add(1).add(0).front()` are two test cases that share the same events with different parameters, which leads to different states (Figure 3).

Relatively, the event's parameter is helpful for distinguishing between states even though any value can be used. Anyhow, the variables with different data types rather than arrays can be easily represented by real numbers. In contrast, the problem arises when representing ar-

rays, as they may have a number of items not a single one. As well, the variables are independent from each other (e.g. *len* can be three while the array may have two items only). So, the applied representation has to consider combinations between items in the array from one side and combinations of the array with the remaining variables from another side.

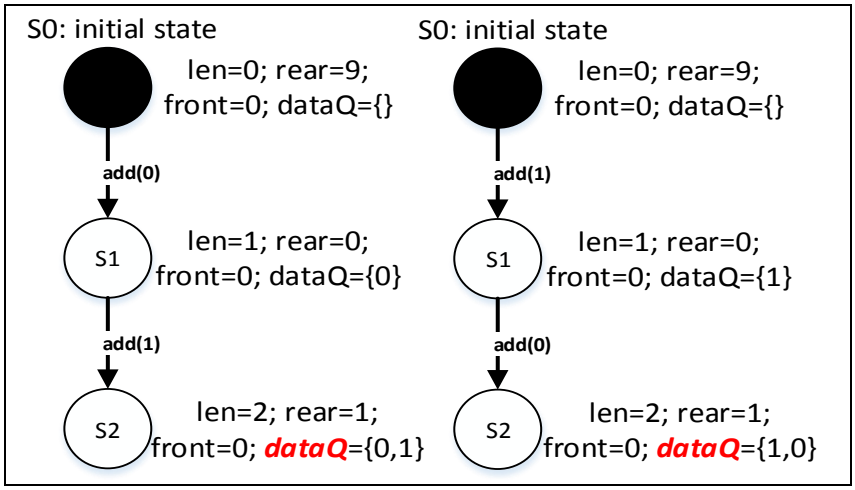

Figure 3. The States for Different Sequences

Therefore, the idea of system ID was used to uniquely represent the combinations in the array. So, a unique integer number is going to be generated randomly in order to replace the values in the array including empty arrays, single values and all possible combinations. The technique continues adding the combinations from the array to a dictionary and assigning a unique value for each combination in order to assure representing all possible combinations in the array. If the combination already covered, it will get the same value stored before. Otherwise, a new item will be added along with a unique identifier. Consequently, all values are going to be grouped altogether and being represented uniquely. Thus, every group will be coded into a real number. Table 3 shows the states representations for `_.add(0).add(1).remove().front()`. The number of digits differs based on the variables and their potential values. For circular queue, the maximum number of digits for dataQ is 10. Besides, *len* can have values up to 10 which needs two digits rather than one as in *rear* and *front* variables, respectively. So, the number of digits for representing a state is going to be 14. Consequently, these states representations are used as an initial population for GA. The chromosome for `_.add(0).add(1).remove().front()` test case is going to be 00901442407170, 0100366712642, 02101028566121, 01111028566121.

Table 3. States Representation

State Number	State Variable				Array Representation	States Representation
	len	rear	front	dataQ		
S0	0	9	0	{}	1442407170	00901442407170
S1	1	0	0	{0}	366712642	0100366712642
S2	2	1	0	{0,1}	1028566121	02101028566121
S3	1	1	1	{0,1}	1028566121	01111028566121

CONCLUSION

We have presented the integration of SSP and GA by representing the states produced by SSP sequence of events in the form that can be accepted by GA. This is a part of an on-going research which aims to enhance the effectiveness of test case generation technique for testing a module with internal memory. We believe that the adoption of GA can improve the effectiveness of SSP to overcome the redundancy issues in SSP and consequently will produce optimized test cases.

REFERENCES

Ali, S., Briand, L. C., Hemmati, H., & Panesar-Walawege, R. K. (2010). A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation. *IEEE Transactions on Software Engineering*, *36*(6), 742-762. doi: 10.1109/TSE.2009.52

Ali, S., Briand, L. C., Hemmati, H., & Panesar-Walawege, R. K. (2010). A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, *36*(6), 742-762.

Alsmadi, I., Alkhateeb, F., Maghayreh, E., Samarah, S., & Doush, I. A. (2010). Effective Generation of Test Cases Using Genetic Algorithms and Optimization Theory. *Journal of Communication and Computer*, *7*(11), 72-82.

Baharom, S., & Shukur, Z. (2008, 26-28 Aug. 2008). Module documentation based testing using Grey-Box approach. Paper presented at *International Symposium on Information Technology 2008*.

Baharom, S., & Shukur, Z. (2010). State-Sensitivity Partitioning Technique for Module Documentation-based Testing. Paper presented at the *Business Transformation through Innovation and Knowledge Management an Academic Perspective*, Istanbul, Turkey.

Baharom, S., & Shukur, Z. (2011). An experimental assessment of module documentation-based testing. *Information and Software Technology*, *53*(7), 747-760. doi: http://dx.doi.org/10.1016/j.infsof.2011.01.005

Conrad, A. P., Roos, R. S., & Kapfhammer, G. M. (2010). *Empirically studying the role of selection operators duringsearch-based test suite prioritization*.

Gannon, J. D., Purtilo, J., & Zelkowitz, M. V. (1994). *Software Specification: A Comparison of Formal Methods*: Ablex Publishing Company.

John, H. (1975). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. USA: University of Michigan.

Kulkarni, N. J., Naveen, K. V., Singh, P., & Srivastava, P. R. (2011). Test Case Optimization Using Artificial Bee Colony Algorithm. *Advances in Computing and Communications*, 570-579.

Li, Z., Harman, M., & Hierons, R. M. (2007). Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering*, *33*(4), 225-237. doi: 10.1109/tse.2007.38

McMinn, P. (2004). Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, *14*(2), 105-156.

Mühlenbein, H., hlenbein, & Schlierkamp-Voosen, D. (1993). Predictive models for the breeder genetic algorithm i. continuous parameter optimization. *Evol. Comput.*, *1*(1), 25-49. doi: 10.1162/evco.1993.1.1.25

Pressman, R. S. (2010). *Software engineering: a practitioner's approach*: McGraw-Hill Higher Education.