# Genetic Algorithm Application for Enhancing State-Sensitivity Partitioning

Ammar Mohammed Sultan, Salmi Baharom(✉),
Abdul Azim Abd Ghani, Jamilah Din, and Hazura Zulzalil

Software Engineering and Information System Department,
Faculty of Computer Science and Information Technology,
Universiti Putra Malaysia, 43400 Serdang, Selangor, Malaysia
ammar.alsultan@hotmail.com,
{salmi,azim,jamilahd,hazura}@upm.edu.my

**Abstract.** Software testing is the most crucial phase in software development life cycle which intends to find faults as much as possible. Test case generation leads the research in software testing. So, many techniques were proposed for the sake of automating the test case generation process. State sensitivity partitioning is a technique that partitions the entire states of a module. The generated test cases are composed of sequences of events. However, there is an infinite set of sequences with no upper bound on the length of a sequence. Thus, a lengthy test sequence might be encountered with redundant data states, which 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 SSP. GA has been identified as the most common potential technique among several optimization techniques. Thus, GA is investigated to integrate it with the existing SSP. This paper addresses the issue on deriving the fitness function for optimizing the sequence of events produced by SSP.

**Keywords:** Genetic Algorithm (GA) · State-Sensitivity partitioning (SSP) · Test case · Sequence of events · Data state

## 1 Introduction

Amongst software development life cycle (SDLC) phases, software testing is the most crucial one [1]. It intends to execute the software and find faults as much as possible. Generally, test case generation dominates the research in software testing while other research areas include test execution and test oracles. Hence, a number of techniques were proposed for improving the effectiveness and efficiency of faults detection. State-sensitivity portioning (SSP) is one of them [2–4].

SSP employs Parnas formal specifications in order to test a module that consists of one or more access programs, which share the same data structure. The output depends on the event triggered, input parameters, conditions and actions. Thus, test data for a module might consist of event sequences (or test sequences) rather than a single event. For the sake of avoiding the exhaustive testing of a module's entire states, SSP partitions the entire states according to their sensitiveness toward events, conditions and

© IFIP International Federation for Information Processing 2015
K. El-Fakih et al. (Eds.): ICTSS 2015, LNCS 9447, pp. 249–256, 2015.
DOI: 10.1007/978-3-319-25945-1_16

actions. Test sequences are selected manually based on all-transitions coverage criterion. However, the sequence of events can be very lengthy and might contain redundant data states, which makes the testing expensive and relatively ineffective.

In the literature, many optimization techniques have been suggested. One technique is search techniques [5, 6] and, among them, genetic algorithm (GA) has been identified as the most common for generating test cases [7]. The success stories of GA inspired us to adopt GA in our work. Similar to other search techniques, the adoption of GA requires the derivation of fitness function [8]. Thus, this paper describes the on-going research that addresses the issue of deriving a fitness function in order to search within the population of states produced by SSP sequence of events. The remainder of this paper is organized as follows: an overview of SSP is presented in the next section; followed by a general overview on GA. Next, the fitness function application in SSP is being described followed by a case study. Finally, the last section summarizes the paper along with the conclusion.

## 2 State-Sensitivity Partitioning (SSP)

A module may consist of one or more access programs that share the same data structure. Its behavior is depending on the event triggered, the value of input parameters and conditions. Consequently, generating test cases for such a module might involve a large number of data states, which grows exponentially in terms of the number of program variables. For example, approximately 10^20 tests (2^32 X 2^32) have to be performed in order to test the correctness of two variables A and B of 32 bit integers, as in [9]. Hence, it would take more than 30,000 years of testing with the assumption of performing 10^8 tests per second. Therefore, it is impossible to explore the space of entire states with limited time resources and memories.

SSP is a test case generation technique for modules [2–4]. The states are partitioned based on state's sensitivity towards events, conditions (pre-conditions) and actions (post-conditions). The goal is to group all states that behave similarly towards access-programs (events), conditions and actions (either sensitive or insensitive) together.

SSP has six sequential steps, which are: (i) identifying sensitive access programs, (ii) partitioning states into equivalence classes, (iii) constructing a state transition model, (iv) selecting test cases based on all-transition coverage criteria, (v) adding the insensitive events to the end of each selected test case and (vi) applying boundary value analysis (BVA) technique in order to select the input parameters. Nonetheless, each test case from the fourth step must be represented by at least one sequence of events. The SSP sequence of events has to be selected randomly as long as it follows the specified conditions of the constructed state transition model in step three (3). Below is an example.

### 2.1 Example

In order to grasp the idea of SSP, let's consider the example of circular queue. Circular queue has three access programs, which are: add(), remove(), and front(). The former

two access programs are sensitive as they modify the data states during their execution while the latter is insensitive as it does not modify the data states. According to SSP, the entire data states are partitioned into equivalence classes based on the number of identified sensitive access programs. So, the circular queue has four possible partitions. In the third step, a state transition model is constructed as in Fig. 1.

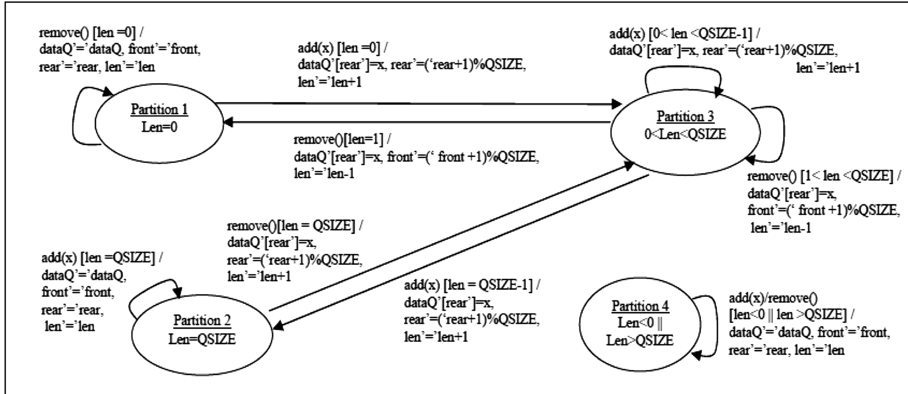


**Fig. 1.** State transition model for circular queue

Once the state transition diagram is constructed, all-transitions coverage criteria will be used for selecting test cases. Table 1 lists the ten test cases obtained from the state transition model. Each test case will be represented by at least one sequence of events. Then, the insensitive events is going to be added to the end of the sequence. Lastly, the BVA technique is applied in order to determine the value of input parameter. With the assumption that maximum length of the circular queue is five, here are some examples of test sequences produced by SSP.

| 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() |

As the SSP sequence of events is selected randomly, any sequence follows the conditions specified by the state transition model is valid. For example, a sequence of events for adding an item to a full queue might include adding twenty items; 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 might be lengthy and contain redundant data states. The lengthy sequence with redundant states makes testing expensive and ineffective. Also, there is a redundancy occurs between two or more test sequences (i.e. sequence of events), where a test sequence is subset from other sequence(s). Therefore, there is a need to optimize the test suites through removing redundant data states. Among the available techniques, search techniques are the most common for obtaining optimized test suites.

**Table 1.** The test cases for circularqueue 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 |

## 3 Genetic Algorithm (GA)

The applications of search techniques in the domain of software testing grew dramatically as they save efforts and times. For test cases generation, GA is the most common amongst all search techniques. It 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 [10]. The typical GA consists of five repetitive steps that continue till the stopping criteria is met. The stopping criteria is either finding an optimum solution or reaching the maximum number of iterations. The GA steps are: (1) random initialization of population that contains candidate solutions. Each solution is represented as a chromosome or sequence of variables [11]; (2) evaluation of new candidate solutions, if the stopping criteria is not met; (3) selection of promising candidate solutions based on fitness function. Fitness function is used for evaluating the solution in terms of its ability to solve the problem; (4) crossover; and (5) mutation.

GA performs search in parallel, which leads to fast calculations. Consequently, software testing leads the GA applications compared with other SDLC phases. This includes different disciplines such as test cases generation [7, 12], test cases prioritization within test suites [13], and test suites reductions [11].

However, prior to apply GA for optimizing the test cases, there is a need to derive the fitness function. 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 search

within for good solutions to be used in GA next iterations. In the next section, the derivation of fitness function is described.

## 4 Fitness Function Application in SSP

Fitness function plays an important role in guiding the search within a population of solutions. It judges whether a potential solution presents a good candidate and, hence, has to be used in GA next iterations. The fitness function comes from existing software metrics followed by several refinements according to the results [8].

Anyhow, SSP sequences of events produced a group of states which are unique and redundant states. In order to optimize SSP, we aim to remove the redundant states. There are two types of redundancies: (1) redundancy in test case level and (2) redundancy in test suite level. Therefore, the calculation of fitness function has to take both types into consideration. We introduce two score namely test case states minimization (TCSM) and test suite states minimization (TSSM). The fitness function is:

$$\text{Fitness} = \text{TCSM} + \text{TSSM} \tag{1}$$

### 4.1 Test Cases States Minimization (TCSM)

TCSM aims to remove redundant states on the test case level, such as the states encountered when trying to add to after reaching the maximum in circular queue or removing when there is no item to be removed. In order to calculate TCSM, there is a need to differentiate between unique and redundant states per sequence of events. A score of TCSM is calculated based on the following equation:

$$\text{TCSM} = \text{USC} + \text{RSC} \tag{2}$$

where USC is the unique states score per sequence of events and RSC is the redundant states score per sequence of events. Let A be a set of unique states in a sequence of events. The calculations for USC is shown in the following equation:

$$USC = \sum_{i=1}^{|A|} \left[\frac{A}{MAX}\right]_i = \frac{1}{MAX} + \frac{1}{MAX} + \ldots + \frac{1}{MAX} \tag{3}$$

where $|A|$ is the cardinality for set A, which counts the number of unique states in the sequence of events and MAX is the maximum number of items that can be added to the data structure. For the calculation of RSC, let B be the set of redundant states in a sequence of events where $B \subseteq A$ and $B \cap A = B$. The calculation for RSC as follows:

$$RSC = \sum_{i=1}^{|B|} \left[\frac{-B}{MAX}\right]_i = -\frac{1}{MAX} - \frac{1}{MAX} - \ldots - \frac{1}{MAX} \tag{4}$$

where |B| is the cardinality for set B, which counts the number of redundant states per sequence of events. Obviously, the score of TCSM can produce a negative value, which indicates that the sequence is unlikely to be in the GA next generations.

### 4.2 Test Suites States Minimization (TSSM)

TSSM focuses on removing redundancies between sequences of events in the test suites. This is due to the fact that some sequences of events are subsets from others. So, for a set C = {$tc_1$, $tc_2$… $tc_n$} of test cases (tc), the TSSM is calculated as follows:

$$\mathrm{TSSM} = \frac{\mathrm{TCO}}{|\mathrm{C}|} \tag{5}$$

where |C| is the cardinality for set C, which counts the number of test cases in the population and TCO is a test case occurrence, which counts the occurrences of a specific sequence within the suite. In order to calculate TCO, every sequence (test case) is considered as an individual set such as: {$tc_1$}, {$tc_2$}, {$tc_n$}. If k is the counter for counting the occurrence of similar test cases, the sets are compared as follows:

$$\forall tc_{n-1} \subseteq tc_n;\ k = k + 1; \tag{6}$$

## 5 Case Study

Assume that the following test suite is produced from the SSP technique based on the circular queue example.

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

To get the fitness, Eq. (1) will be used. However, the values may be greater than or equal to one. So, there is a need to use the average fitness as follows:

$$Average\ Fitness = \frac{Fitness}{Total\ Fitness} \tag{7}$$

where Total Fitness is the summation of all fitness values in the population. Table 2 shows the fitness along with the average fitness for the population above.

The results show that test cases with events close to the maximum number of items that can be added to the data structure. Hence, TC3 got the highest value followed by

**Table 2.** The fitness values

| ID | SC | TSM | Fitness | Average fitness |
|---|---|---|---|---|
| TC1 | 0.2 | 0.4 | 0.6 | 0.15 |
| TC2 | 0.2 | 0.4 | 0.6 | 0.15 |
| TC3 | 1.2 | 0.2 | 1.4 | 0.35 |
| TC4 | 1 | 0.2 | 1.2 | 0.3 |
| TC5 | 0 | 0.2 | 0.2 | 0.05 |
| TOTAL | | | 4 | 1 |

TC4. Besides, the test cases with redundant events, such as TC5, obtain the lowest value.

## 6 Conclusion

The integration of SSP and GA is promising in order to optimize sequence of events. Prior to any application, there is a need to derive a fitness function that guides the search for solutions within a population. 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

1. Pressman, R.S.: Software Engineering: A Practitioner's Approach. McGraw-Hill Higher Education, New York (2010)
2. Baharom, S., Shukur, Z.: Module documentation based testing using Grey-Box approach. In: ITSim 2008. International Symposium on Information Technology, 2008 (2008)
3. Baharom, S., Shukur, Z.: State-Sensitivity Partitioning technique for module documentation-based testing. In: Business Transformation through Innovation and Knowledge Management an Academic Perspective. Istanbul, Turkey (2010)
4. Baharom, S., Shukur, Z.: An experimental assessment of module documentation-based testing. Inf. Softw. Technol. **53**(7), 747–760 (2011)
5. Alsmadi, I., et al.: Effective generation of test cases using genetic algorithms and optimization theory. J. Commun. Comput. **7**(11), 72–82 (2010)
6. Kulkarni, N.J., et al.: Test case optimization using artificial bee colony algorithm. In: Abraham, A., Mauri, J.L., Buford, J.F., Suzuki, J., Thampi, S.M. (eds.) Advances in Computing and Communications. Communications in Computer and Information Science, vol. 192, pp. 570–579. Springer, Heidelberg (2011)
7. Ali, S., et al.: A systematic review of the application and empirical investigation of search-based test case generation. IEEE Trans. Softw. Eng. **36**(6), 742–762 (2010)
8. Harman, M., McMinn, P., de Souza, J.T., Yoo, S.: Search based software engineering: techniques, taxonomy, tutorial. In: Meyer, B., Nordio, M. (eds.) Empirical Software Engineering and Verification. LNCS, vol. 7007, pp. 1–59. Springer, Heidelberg (2012)

9. Gannon, J.D., Purtilo, J., Zelkowitz, M.V.: Software Specification: A Comparison of Formal Methods. Ablex Publishing Company, Norwood (1994)
10. Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. University of Michigan, Ann Arbor (1975)
11. Li, Z., Harman, M., Hierons, R.M.: Search algorithms for regression test case prioritization. IEEE Trans. Softw. Eng. **33**(4), 225–237 (2007)
12. McMinn, P.: Search-based software test data generation: a survey. Softw. Test. Verification Reliab. **14**(2), 105–156 (2004)
13. Conrad, A.P., Roos, R.S., Kapfhammer, G.M.: Empirically studying the role of selection operators duringsearch-based test suite prioritization. ACM (2010)