Reducing Test Suite of State-Sensitivity Partitioning (SSP)

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

***Abstract*—Software testing is one of the most vital phases of software development lifecycle that aims to detect software faults. Test case generation dominates the software testing research. SSP is one of many techniques proposed for test case generation. The goal of SSP is to avoid exhaustively testing all possible combinations of inputs and preconditions. The test cases produced by SSP are formed of a sequence of events. For instance, a queue test case might encompass the addition of thirty items onto the queue; deletion of three items, addition of sixty more items, seven deletions and examining the outcome. Notwithstanding perceiving the finite bounds of the queue size, there is an endless engage of sequences along with no upper limit on the sequence's length. Therefore, the sequence might get lengthy as a result of comprising data states that are redundant. The test suite size is expanded due to the data states redundancies and subsequently, the testing process will become insufficient. Thus, it is a necessity to optimize the SSP test suite by removing the redundant data states. This paper addresses the issue of SSP suite reduction, which part of the process for optimizing test suite produced by the SSP.**

***Index Terms*—Data State; State-Sensitivity Partitioning (SSP); Sequence Of Events; Test Case; Test Suite Reduction.**

I. INTRODUCTION

Software testing is one of the most vital phases of software development lifecycle (SDLC) [1]. It aims to detect as much as possible of software faults through executing the software against a number of test cases with a specific objective. Within the testing research, test case generation dominates due to test cases' principality for testing. However, the classical test case generation technique relies on examining all possible combinations of inputs and preconditions. Consequently, a number of techniques were proposed in order to overcome this exhaustive testing which is not applicable for all systems. One of these techniques is State-sensitivity partitioning (SSP) [2-4].

SSP is a module-based technique for generating test cases [2-4]. It determines the sensitiveness of states towards events, conditions and actions and, hence, partitions the states accordingly. The objective is to gather all data states that behave on likewise towards access-programs (events), conditions sensitive actions and insensitive actions altogether in a group. For SSP, six steps have to be conducted consecutively, which are: (i) identification of sensitive access programs, (ii) partitioning of states into equivalence classes, (iii) construction of a state transition model, (iv) selection of test cases using all-transition coverage criteria, (v) ending every test case that was selected with an insensitive event and (vi) selection of the input parameters based on boundary value analysis (BVA) technique. Nonetheless, each test case from the fourth step (iv) has to be represented by one sequence of events at least. SSP sequences of events are selected randomly following the conditions specified in the state transition model that was constructed in the third step (iii). For instance, a queue test case that tests the queue's behavior towards adding items to a full queue might include the addition of thirty items onto the queue; deletion of nineteen items, addition of sixty more, deletion of sixty-four, the addition of twenty more, deletion twenty, two more additions and examining the outcome. Consequently, a lengthy sequence of events might be generated with a number of redundant data states, which clearly increases testing expenses and makes testing ineffective.

In a previous work [5], we optimized SSP sequence of events by adopting a search technique, genetic algorithm (GA). This is inspired by the fact that GA has been acknowledged as the most common search technique intended for test cases generation [6]. Although the application of GA shows a significant reduction in the redundancies issues, it missed some redundancies within a test suite. Such redundancies occurred because of the SSP selecting test cases (i.e. sequences of events) based on all-transition coverage criteria. Therefore, the test suite produced by the SSP consists of several categories of sequences of events. The adoption of GA has successfully removed redundancies of test suite within the same category. However, it ignores redundancies between different categories. Hence, this paper depicts the on-going research that addresses the issue of removing redundancies in the test suite, between categories in particular. It is so-called SSP suite reduction. The remainder of this paper is organized as follows: the next section presents an outline of SSP; followed by a general overview of the test suite reduction. Next, a case study is demonstrated and the SSP suite reduction is being described. Finally, the closing section concludes the paper alongside with a summary.

II. STATE SENSITIVITY PARTITIONING TECHNIQUE

For understanding the idea of SSP, let's take optimum request order as an example. It has three access programs, which are: request(), delRequest(), and calculate(). Both request() and delRequest() access programs are considered sensitive as they modify the internal data states during their executions. On the other hand, calculate() access program is insensitive as it does not affect the internal data states. Consequently, the optimum request order has four possible partitions. The SSP technique partitions the entire data states

into equivalence classes in accordance with the number of identified sensitive access programs. Figure 1 presents a state transition model of optimum request order based on the identified equivalence classes.

Following the construction of the state transition model, the next step is to select test cases using all-transitions coverage criteria. Ten categories of test cases (i.e. sequences of events) listed in Table 1 as obtained from the state transition model.

Each test case from the table will be represented by one test data, at least, that is formed of sequences of events. The sequence of events then will be added along with insensitive events to the end of the sequence. Lastly, the boundary value analysis (BVA) technique is applied in order for determining the input parameter values.

Assuming the maximum length of the optimum request order program is five, Table 2 shows a number of examples of test sequences produced by SSP technique.

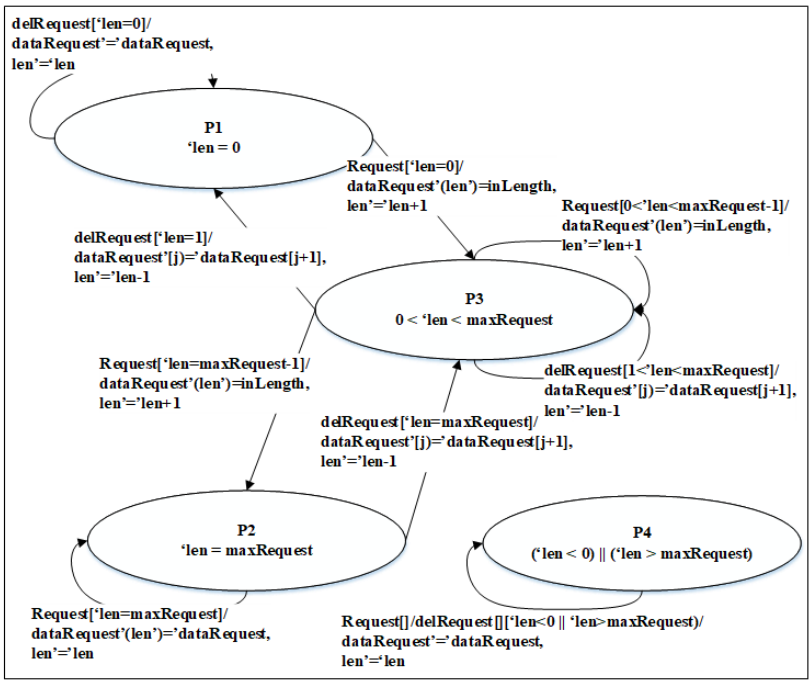

Figure 1: State Transition Model for Optimum Request Order Program

Table 1
The Test Cases for Optimum Request Order program

Category	P	Event	Pre-condition	Post-condition
1	1	request	len=0	dataRequest'[len',0]=inLength,dataRequest'[len',1]=inQty,len'='len+1
2	1	delRequest	len=0	dataRequest'[len',0]='dataRequest,dataRequest'[len',1]='dataRequest, len'='len
3	2	request	len=maxRequest	dataRequest'[len',0]='dataRequest,dataRequest'[len',1]='dataRequest, len'='len
4	2	delRequest	len=maxRequest	(∀j:indexOf(inLength)≤j<len', dataRequest'[j,0] ='dataRequest[j + 1, 0]), len'='len–1
5	3	Request	0<len<maxREquest – 1	dataRequest'[len',0]=inLength,dataRequest'[len',1]=inQty,len'='len+1
6	3	request	len=maxRequest – 1	dataRequest'[len',0]=inLength,dataRequest'[len',1]=inQty,len'='len+1
7	3	delRequest	1<len<maxRequest	(∀j:indexOf(inLength)≤j<len', dataRequest'[j,0] ='dataRequest[j + 1, 0]), len'='len–1
8	3	delRequest	len=1	(∀j:indexOf(inLength)≤j<len', dataRequest'[j,0] ='dataRequest[j + 1, 0]), len'='len–1
9	4	request	len<0 && len>maxRequest	dataRequest'[len',0]='dataRequest,dataRequest'[len',1]='dataRequest, len'='len
10	4	delRequest	len<0 && len>maxRequest	dataRequest'[len',0]='dataRequest,dataRequest'[len',1]='dataRequest, len'='len

Table 2
Examples of Optimum Request Order Test cases

Category	TC#	Sequence of events
1	TC1	_.request(0,0).calculate()
	TC2	_.request(1,1).calculate()
	TC3	_.request(-1,-1).calculate()
	TC4	_.request(1295644148, 1295644148).calculate()
	TC5	_.request(-1295644148,- 1295644148).calculate()
	TC6	_.request(Integer.Max_value, Integer.Max_value).calculate()
	TC7	_.request(Integer.Min_value, Integer.Min_value).calculate()
2	TC8	_.delRequest().calculate()
	TC9	_.request(0,0).request(1,1).delRequest(0).request(-1,-1).request(1295644148,1295644148). request(-1295644148,-1295644148).request (Integer.Max,Integer.Max).request(Integer.Min,Integer.Min).calculate()
	TC10	_.request(1,1).request(-1,-1).delRequest(1) .request(1295644148, 1295644148).request(-1295644148,-295644148).request(Integer.Max, Integer.Max).request(Integer.Min,Integer.Min) .request(0,0).calculate()
	TC11	_.request(-1,-1).request(1295644148,12956441 48).delRequest(-1).request(-1295644148,-1295 644148).request(Integer.Max,Integer.Max).request(Integer.Min,Integer.Min).request(0,0).request(1,1).calculate()
3	TC12	_.request(1295644148,1295644148).request(-12 95644148,-1295644148).delRequest(12 95644148).request(Integer.Max,Integer.Max).request(Integer.Min,Integer.Min).request(0,0).request(1,1).request(-1,-1).calculate()
	TC13	_.request(-1295644148,-1295644148).reques t(Integer.Max,Integer.Max).delRequest(-1295 644148).request(Integer.Min,Integer.Min).req uest(0,0).request(1,1).request(-1,-1).request (1295644148,1295644148).calculate()
	TC14	_.request(Integer.Max,Integer.Max).request (Integer.Min,Integer.Min).delRequest(Integer .Max).request(0,0).request(1,1).request(-1,-1) .request(1295644148,1295644148).request(-1295644148,-1295644148).calculate()
	TC15	_.request(Integer.Min,Integer.Min).request(0,0).delRequest(Integer.Min).request(1,1).request(-1,-1).request(1295644148,1295644148).re quest(-1295644148,-1295644148).request (Integer.Max,Integer.Max).calculate()

Category	TC#	Sequence of events
9	TC16	_.delRequest(0).request(0,0).calculate()
	TC17	_.delRequest(1).request(1,1).calculate()
	TC18	_.delRequest(-1).request(-1,-1).calculate()
	TC19	_.delRequest(1295644148).request(1295644148,1295644148).calculate()
	TC20	_.delRequest(-1295644148).request(-1295644148,- 1295644148) .calculate()
	TC21	_.delRequest(Integer.Max_value).request(Integer.Max_value,Integer.Max_value).calculate()
	TC22	_.delRequest(Integer.Min_value).request(Integer.Min_value,Integer.Min_value).calculate()
	TC23	_.request(0,0).request(1,1).request(-1,-1) .request(1295644148,1295644148).request(-1295644148,-1295644148).request(Integer .Max,Integer.Max).request(Integer.Min, Integer.Min).request(0,0).calculate()
	TC24	_.request(1,1).request(-1,-1).request(1295644 148,1295644148).request(-1295644148,-12956 44148).request(Integer.Max,Integer.Max) .request(Integer.Min,Integer.Min).request(0,0).request(1,1).calculate()
	TC25	_.request(-1,-1).request(1295644148,1295644 148).request(-1295644148,-1295644148).re quest(Integer.Max,Integer.Max).request(Integer.Min,Integer.Min).request(0,0).request(1,1).request(-1,-1).calculate()
	TC26	_.request(1295644148,1295644148).request(-1295644148,-1295644148).request (Integer. Max,Integer.Max).request(Integer.Min,Integer .Min).request(0,0).request(1,1).request(-1,-1) .request(1295644148,1295644148).calculate()
	TC27	_.request(-1295644148,1295644148).request(Integer.Max,Integer.Max).request(Integer.Min,Integer.Min).request(0,0).request(1,1).requst(-1,-1).request(1295644148,1295644 148).r equest(-1295644148,-1295644148).calculate()
	TC28	_.request(Integer.Max,Integer.Max).request(Integer.Min,Integer.Min).request(0,0).request(1,1).request(-1,-1).request(1295644148,12956441 48).request(-1295644148,-1295644148).req uest(Integer.Max,Integer.Max).calculate()
	TC29	_.request(Integer.Min,Integer.Min).request(0,0).request(1,1).request(-1,-1).request(12956441 48,1295644148).request(-1295644148,-1295644148).request(Integer.Max,Integer.Max).request(Integer.Min,Integer.Min).calculate()
10	TC30	_.delRequest(0).delRequest(1).calculate()
	TC31	_.request(0,0).request(1,1).request(-1,-1).request(1295644148,1295644148).request(-1295644148,-1295644148).request(Integer .Max, Integer.Max).delRequest(0).calculate()
	TC32	_.request(1,1).request(-1,-1).request(1295644 148,1295644148).request(-1295644148,-1295644148).request(Integer.Max,Integer.Max).request(Integer.Min,Integer.Min).delRequest(1).calculate()
	TC33	_.request(-1,-1).request(1295644148,1295644 148).request(-1295644148,-1295644148).req uest(Integer.Max,Integer.Max).request(Integer.Min,Integer.Min).request(0,0).delRequest(-1).calculate()
	TC34	_.request(1295644148,1295644148).request(-1295644148,-1295644148).request(Integer .Max,Integer.Max).request(Integer.Min, Integer.Min).request(0,0).request(1,1).delRequest(1295644148).calculate()
	TC35	_.request(-1295644148,-1295644148).reque st(Integer.Max,Integer.Max).request(Integer.Min,Integer.Min).request(0,0).request(1,1) .request(-1,-1).delRequest(-1295644148).calculate()
	TC36	_.request(Integer.Max,Integer.Max).request(Integer.Min,Integer.Min).request(0,0).request(1,1).request(-1,-1).request(1295644148,129 5644148).delRequest(Integer.Max).calculate()
	TC37	_.request(Integer.Min,Integer.Min).request(0,0).request(1,1).request(-1,-1).request(129564 4148,1295644148).request(-1295644148,-129 5644148).delRequest(Integer.Max).calculate()

SSP generated sequence of events can be very lengthy as they are being selected randomly, which is valid as long as it follows the specified conditions of the constructed state transition model. A lengthy sequence of events might contain redundant data states within a sequence. Besides, another redundancy may occur between two or more test cases (i.e. sequences of events), where one sequence appears as a subset from another test sequence(s). Axiomatically, a test suite with such redundancies makes the testing expensive and quite ineffective. Therefore, there is a need to find an appropriate technique for optimizing such test suites throughout removing redundant data states. One of the available techniques includes using search techniques, which are the most common for obtaining optimized test suites.

III. GENETIC ALGORITHM

According to [7], genetic algorithm (GA) is the most common search technique for software testing. Moreover, GA was highly employed for test case generation with the purpose of producing optimized test cases [6]. It mimics the natural evolution theory by Darwin [8]. The typical GA, shown in Figure 2, is comprised of five steps [9]: i) initializing a random population, (ii) evaluation of solutions, (iii) selection of candidates based on a fitness function, (iv) crossover and (v) mutation. Within a potential population of solutions, fitness function guides the search. Crossover is the process of exchanging the genes between two chromosomes from some random point, while mutation stands for replacing a random gene in any chromosome with available values.

The application of GA on the test cases that are composed of sequences of events shows a reduction of states redundancies within a sequence along with representing each category by a single sequence. In SSP, there is a need to preserve the categories for assuring that all-transitions are covered. The following Table 3 shows the results of GA application in optimum request order example for five selected categories.

The results show a significant reduction in the number of the sequence of events for each category. However, the reduction based on GA missed redundancies that occur among those categories. As a result, there is a need to employ a technique to eliminate such redundancies. The following section outlines this technique.

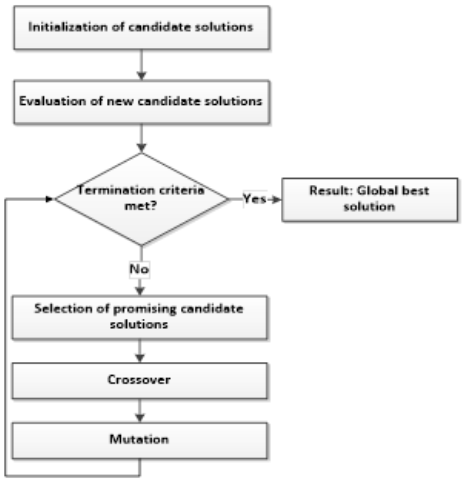

Figure 2: Workflow of GA

Table 3
Results of GA Application

Category	TC#	Sequence of events
1	TC1	_.request(0,0).calculate()
2	TC8	_.delRequest(0).calculate()
3	TC9	_.request(0,0).request(1,1).delRequest(0).req uest(-1,-1).request(1295644148,1295644148) .request(-1295644148,-1295644148).request (Integer.Max,Integer.Max).request(Integer.Min,Integer.Min).calculate()
9	TC16	_.delRequest(0).request(0,0).calculate()
	TC23	_.request(0,0).request(1,1).request(-1,-1) .request(1295644148,1295644148).request(-1295644148,-1295644148).request(Integer .Max,Integer.Max).request(Integer.Min, Integer.Min).request(0,0).calculate()
10	TC30	_.delRequest(0).delRequest(1).calculate()
	TC31	_.request(0,0).request(1,1).request(-1,-1).request(1295644148,1295644148).request(-1295644148,-1295644148).request(Integer .Max, Integer.Max).delRequest(0).calculate()

IV. SSP SUITE REDUCTION

With respect to the test coverage, a number of suite reduction techniques had been proposed such as selecting test cases that satisfy most of the requirements [10]. Another technique considered finding a minimal representative set from the suite using Quinn-McCluskey [11]. Fuzzy logic was employed by [12] for reducing test suites. Similarly, [13] proposed a metric to be applied on a test suite for eliminating similarities that can be detected among test cases. However, these techniques did not consider redundancies within the whole suite rather than redundancies per sequence.

For test sequences that are subset of other sequences and generate the same data states are considered redundant and, hence, one of them should be eliminated. Referring to Table 3, we can eliminate TC1 while maintaining the suite coverage as TC9 has covered similar data state. We propose a technique namely SSP Suite Reduction to eliminate such redundancy. The reduction algorithm is shown in Figure 3 while Table 4 shows the results of SSP Suite Reduction for the datasets shown in Table 3.

```
1. T = [T1,T2, T3, T4,..... Tn]
2. len = T.len;
3. x= 0;                    //first counter
4. y = x + 1;               //second counter
5.   for (t =0; t < len ; t++){
6.    a=T[x].len<= T[y].len?T[x]:T[y];
7.    b=T[x].len<= T[y].len?T[y]:T[x];
8.    if (identical(a, b){
9.          T.remove(a);
10.         if (T[x]==a){
11.             y = x + 1;
12.         } else {
13.             x = x + 1;
14.         }
15.         len = T.len;
16.   } else {
17.      y = y + 1;
18.   }
19. }
```

```
identical(T1, T2) {
  j=0
  for (i=0; i<T2.len; i++){
      if (T2[i]== T1[j]) {
          //true
          j++;
      } else {
          j=0;
      }

    if (j == T1.len -1) {
      return true;
    }
  }
  return false;
}
```

Figure 3: SSP Suite Reduction Algorithm

As in the algorithm, T is a test suite with a number of test cases that determines its length (len). Two counters are used to go through the suite. Next, two variables (a and b) are used to store the sequences to be compared. Subsequently, "identical" Boolean function is called to compare the sequences considering that the shorter sequence is potentially subset from the other one. If the sequences were found in other ones in the same order, it will be considered subset and the function will return true. Hence, being removed in "remove" function. This continues till going through the whole suite.

Table 4
Results of SSP Suite Reduction

Category	TC#	Sequence of events
3	TC9	_.request(0,0).request(1,1).delRequest(0).req uest(-1,-1).request(1295644148,129564 4148) .request(-1295644148,-1295644148).request (Integer.Max,Integer.Max).request(Integer.Min,Integer.Min).calculate()
9	TC16	_.delRequest(0).request(0,0).calculate()
	TC23	_.request(0,0).request(1,1).request(-1,-1) .request(1295644148,1295644148).request(-1295644148,-1295644148).request(Integer .Max,Integer.Max).request(Integer.Min, Integer.Min).request(0,0).calculate()
10	TC30	_.delRequest(0).delRequest(1).calculate()
	TC31	_.request(0,0).request(1,1).request(-1,-1).request(1295644148,1295644148).request(-1295644148,-1295644148).request (Integer.Max, Integer.Max).delRequest(0) .calculate()

In this table, category 1 (TC1) was removed as it was found subset from category 3 (TC9), category 9 (TC23) and category 10 (TC31). Besides, category 2 (TC8) was eliminated as it was subset of category 9 (TC16) and category 10 (TC30).

The percentage of suite size reduction (SSR) metric is employed [14] to evaluate the results. A test suite T with a number of test cases |T| and a reduced suite Tred with a number of test cases |Tred| SSR is calculated as follows:

$$SSR = \frac{|T| - |T_{red}|}{|T|} * 100\% \quad (1)$$

V. CASE STUDY

For simplicity, let's consider another example highlighting the reduction procedure. Tic-Tac-Toe is a one-to-one game where a board of 3x3 is occupied by either 'X' or 'O' marks. Hence, it has two events: play(x) and undo(). The former specifies the cell to be filled while the latter clears the last occupied cell.

Using SSP, the main categories for Tic-Tac-Toe are shown in Table 5. It results in 101 sequences using the specified data.

In the next phase, GA is applied on all sequences per category intending to reduce redundant states per category. Table 6 outlines the GA parameters to be used for running the application.

Since the GA reduced suite still include a number of redundant sequences (i.e. two or more sequences that generate identical states where one of them is subset from others), there is a need to eliminate such redundancies. Hence, the suite reduction algorithm has to be applied and the results will be similar to Table 7.

Table 5
Categories of Tic-Tac-Toe program

Category	Format of sequence of events
1	_.play(x)
2	_.undo()
3	_.play(x).play(x).undo().play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x)
4	_.play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).undo()
5	_.play(x).play(x).undo().play(x).play(x)
6	_.play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x)
7	_.play(x).play(x).undo()
8	_.play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).undo()
9	_.play(x).undo()
10	_.undo().play(x)
11	_.play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x)
12	_.undo().undo()
13	_.play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).undo()

Table 6
GA parameters

Parameters	Values
Population size	10
Number of generations	1000
Mutation rate	0.1
Crossover rate	0.3
Termination criteria	Generations = 1000 or lowering the fitness value

Table 7
Categories of reduced Tic-Tac-Toe

Category	Format of sequence of events
3	_.play(x).play(x).undo().play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x)
4	_.play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).undo()
8	_.play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).undo()
9	_.play(x).undo()
10	_.undo().play(x)
11	_.play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x)
12	_.undo().undo()
13	_.play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).play(x).undo()

VI. Results and Discussion

Table 8 shows the reduction of each category in the Optimum Request Order program while Table 9 shows the reduction of each category in Tic-Tac-Toe program.

In this table, a category refers to a group of sequences that have the same structure but differ in their parameters. Consequently, each optimum request order category has 7 sequences except in category 2, category 7, category 9 and category 10. For category 2, there is no parameter which results only in one sequence. Category 7 and category 9 have 14 sequences due to having 2 scenarios, while category 10 has 2 conditions, len <0 and len >maxRequest, which result in 8 sequences for both events.

Table 8
SSR for Optimum Request Order categories

Category	|TSSP|	|TGA|	|TSSPRed|
1	7	1	0
2	7	1	0
3	7	1	1
4	7	1	1
5	7	1	0
6	7	1	0
7	14	2	1
8	7	1	1
9	14	2	1
10	14	2	1
Total	91	13	8
	SSR		91.21%

Table 9
SSR for Tic-Tac-Toe categories

Category	|TSSP|	|TGA|	|TSSPRed|
1	9	1	0
2	1	1	0
3	9	1	1
4	9	1	1
5	9	1	0
6	9	1	0
7	18	2	1
8	9	1	1
9	18	2	1
10	10	2	1
Total	101	13	8
	SSR		92.1%

On the other hand, each Tic-Tac-Toe category has 9 sequences as a result of having 9 different cells to be occupied except in category 2, category 7, category 9 and category 10 which have 1, 18, 18 and 10 sequences respectively.

Whilst applying GA on the datasets result reduced the sequences per category featuring redundant states per sequence, the suite reduction algorithm eliminated subset categories.

The results outline that SSP suite reduction complements the GA reduction for reducing the redundancies within a suite. Some categories were eliminated as a result of being covered by other categories, such as category 1 is subset of category 3. We also conducted an experimental study using mutation analysis to evaluate the effectiveness of SSP Suite Reduction. The experiment shows that the SSP Suite Reduction (i.e. 8 test cases) has killed the same number of mutants in comparison to SSP (i.e. 91 test cases in optimum request order and 101 test cases in Tic-Tac-Toe).

VII. Conclusion

This paper presents a part of an on-going research which intends to enhance the effectiveness of test case generation technique for testing a module with internal memory. We believe that incorporating SSP suite reduction can improve the effectiveness of SSP in detecting faults.

VIII. Acknowledgement

The authors would like to acknowledge The Ministry of Higher Education Malaysia (MOHE) for the financial support of this research. This research is supported by MOHE under the Fundamental Research Grant Scheme (FRGS) with project code: 08-01-15-1723FR.

References

[1] Pressman, R. and B. Maxim, Software Engineering: A Practitioner's Approach. 2014: McGraw-Hill Education.
[2] Baharom, S. and Z. Shukur. Module documentation based testing using Grey-Box approach. in ITSim 2008. International Symposium on Information Technology, 2008. 2008.
[3] Baharom, S. and Z. Shukur. State-Sensitivity Partitioning Technique for Module Documentation-based Testing. in Business Transformation through Innovation and Knowledge Management An Academic Perspective. 2010. Istanbul, Turkey.
[4] Baharom, S. and Z. Shukur, An experimental assessment of module documentation-based testing. Information and Software Technology, 2011. 53(7): p. 747-760.
[5] Sultan, A.M., et al., Genetic Algorithm Application for Enhancing State-Sensitivity Partitioning, in Testing Software and Systems: 27th IFIP WG 6.1 International Conference, ICTSS 2015, Sharjah and Dubai, United Arab Emirates, November 23-25, 2015, Proceedings, K. El-Fakih, G. Barlas, and N. Yevtushenko, Editors. 2015, Springer International Publishing: Cham. p. 249-256.
[6] Ali, S., et al., A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation. IEEE Transactions on Software Engineering, 2010. 36(6): p. 742-762.
[7] Harman, M., S.A. Mansouri, and Y. Zhang, Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys (CSUR), 2012. 45(1): p. 11.
[8] Chong, E.K.P. and S.H. Zak, An Introduction to Optimization. 2013: Wiley.
[9] Boussaïd, I., J. Lepagnot, and P. Siarry, A survey on optimization metaheuristics. Information Sciences, 2013. 237: p. 82-117.
[10] Parsa, S. and A. Khalilian, On the optimization approach towards test suite minimization. International Journal of Software Engineering and its applications, 2010. 4(1): p. 15-28.
[11] Zhang, R., et al. A New Method for Test Suite Reduction. in The 9th International Conference for Young Computer Scientists, 2008. ICYCS 2008. 2008.
[12] Huang, C.-Y., C.-S. Chen, and C.-E. Lai, Evaluation and analysis of incorporating Fuzzy Expert System approach into test suite reduction. Information and Software Technology, 2016. 79: p. 79-105.
[13] Sapaat, M.A. and S. Baharom. A preliminary investigation towards test suite optimization approach for enhanced State-Sensitivity Partitioning. in 2nd International Conference on Instrumentation, Communications, Information Technology, and Biomedical Engineering (ICICI-BME), 2011. 2011.
[14] Shi, A., et al., Balancing trade-offs in test-suite reduction, in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014, ACM: Hong Kong, China. p. 246-256.