Improving Code Coverage for Structural Testing by using Parametric Inter-procedural Slicing

Myint Myitzu Aung
Software Engineering Lab
University of Computer Studies, Mandalay
Mandalay, Myanmar
myintmyitzuaung@ucsm.edu.mm

Kay Thi Win
Faculty of Information Science
University of Computer Studies, Mandalay
Mandalay, Myanmar
kaythiwin@ucsm.edu.mm

***Abstract*— Structural testing is one of the testing techniques in software testing. It is the testing process of internal structure in the given source code by comparing the expected result and real results and finding out faults. To complete the structural testing, it takes a long time and needs to improve code coverage. Thus, this paper focus on improving code coverage of structural testing by using parametric inter-procedural slicing with backward slicing technique. In program slicing, there are two types of slicing: forward and backward slicing depending on the slicing direction. In this paper, backward slicing is used and it starts at the criteria for the slice consideration. The parametric inter-procedural slicing is one of the program slicing techniques which is an algorithm that takes the slice criteria as a parameter and work to handle inter-procedural data flow by considering many possible dependence relations. By slicing the original input program before measuring code coverage, the input program is simplified and excluded unexecutable statements. Therefore, the proposed system in this paper can reduce the complexity of the source code and improve code coverage for structural testing. Moreover, the used dataset in experiment is benchmark dataset in C and it is converted to Java for the improvement of language oriented.**

Keywords— Structural testing, parametric inter-procedural slicing, backward slicing.

I. INTRODUCTION

In software testing, there are two approaches to perform. They are structural testing and functional testing. The structural testing is mainly focused on the internal structure of the source code program and the functional testing is also focused on the functional structure. In structural testing, it compares the actual and estimated result and it has to find out faults. Therefore it is a technique to evaluate a system by comparing its results (estimated and actual) result automatically or manually. To perform the structural testing absolutely and completely, it is very hard to be possible because it is a time consuming task. But, program slicing can overcome this. Program slicing is a significant testing technique because it can understand the structure of the program and software which are decomposed into smaller units based on the different dependencies (control, method call, data etc) between the program statements [1]. By using the program slicing, every slice includes the statements which are related to a specific variable and ignore other statements [15]. Thus, the original program becomes simplified program and it is reduced complexity by using this program slicing technique [2].

Most recent researches on structural testing have based on improving code coverage and there is needed to improve quality that is to improve code coverage of excluding the unexecutable statements [11]. In this paper, a proposed system is presented to improve code coverage in Section 4. Sections 2 and 3 are related work and background theory.

II. RELATED WORKS

There are several methods for improving different coverage has been proposed in the past. Most of the techniques are used in several testing techniques that require the understanding of the internal working of the program. By using these techniques, there are many improving in coverage, accuracy and solving in some problems but there is still some complexity and performance in measuring test case coverage [16].

Sakti, Pesant, and Gueheneuc studied generating the appropriate sequences of method calls and they proposed an approach to construct the test-data generation problem in unit-class testing. The implementation of this proposed approach is JTExpert. It is very effective in searching test-data suite and it can provide high code coverage 70% in only less than 10s. This paper presented almost hundred classes which are taken from various Java libraries of open-sources and the code coverage of JTExpert is higher than EvoSuite but it needs less time. The JTExpert supports branch coverage and it is based on a guided random search [4]. Moreover, the authors were still trying to add other search algorithms, such as hill climbing and genetic algorithm.

Alatawi, Miller and Søndergaard proposed metamorphic testing. This is a testing technique in which domain-specific properties about program behavior are used and it relates pairs of inputs to pairs of outputs. The inputs are generated in the form of follow-up tests and they compared their outputs with the original output with the help of metamorphic relations. Their assumption is that by the use of metamorphic testing, the capacity of a Dynamic Symbolic Execution (DSE) test suite to uncover faults and these follow-up tests achieve uncovered segments has increased [5]. They have tested seven nontrivial libraries, compared two types of test suites from DSE+MT and DSE, presenting the improving coverage, and finding more faults. They may be difficult in handing the large-scale program.

Wong et. al. proposed Document-Assisted Symbolic Execution (DASE) for the purpose of getting automated test

generation and better bug detection. To take out input constraints, DASE leverages heuristics in NLP techniques to explore program documentation. To evaluate DASE, they focused on about 88 programs in 5 software suites which are used in real-life. Without input constraint, symbolic execution can fail to identify but it can detect 12 unknown bugs in which 6 bugs are confirmed by some developers [6]. Moreover, DASE can increase branch coverage, line coverage and call coverage. But there is a difficulty to against the input to base on testing error handling code.

Harman, Jia and Zhang presented an analysis of their research agenda called Search Based Energy Testing (SBET). For Test Strategy Identification, they descripted Multi-objective SBST and SBST [7]. The authors concluded with a presenting the about of FIFIVERIFY tool, which can identify faults and bugs, handle them and validate the fixes automatically.

Satyam proposed an automated technique that appears as a promising technique to eliminate test time and effort. The author used a code transformation technique. The input is simple java program and it is transformed by using four algorithms. The author used Quine Mc-cluskey method and Petric methods to achieve the transformed program. After transforming the program, a tool called Cobertura provides the branch coverage of that transformed program. The measurement of branch coverage using this transformed program is higher than the coverage of original program [8]. The technique is to increase in branch coverage that is compared with traditional techniques.

Sangeetha, and Ramasundaram proposed a new tool to give optimality. This tool provides the effectiveness that is not affected by infeasible elements in the source code. They use Eclipse IDE, Java Swing and java to implement this tool [9]. This paper deals with various testing techniques some testing tools.

The authors in [13] presented that there are three problems. First, path coverage may not be adapted to the criteria under consideration. Second, some of the coverage requirements may be infeasible. Third, the authors need to be able to handle a large range of different coverage criteria. They proposed three ingredients to tame these issues: a unified management of a large class of coverage criteria through labels (i.e. reachability objectives), a variant of DSE designed to handle explicit coverage requirements at only a reasonable cost, and a combination of well-known static analyses for detecting infeasible coverage requirements. These results have been implemented into the LTest plugin of the open-source software analyzer Frama-C. The authors also presented new results, including experiments on a weak form of the MCDC (or Mutations) criterion and a combination of DSE with infeasibility detection [13].

The authors in [14] used one of the mining methods, K-Mean Algorithm in order to reduce the number of test suites thereby facilitating the mining from test cases. These researchers have implemented the K-Means Algorithm for partitioning the test suite of sample module or the input domain into an expected number (K) of partitions. Moreover, the authors described that the defined code coverage criteria indicate test suite reduction in the purpose of selecting a test suite that will give 100% code coverage [14].

III. BACKGROUND

The program slicing is a program analysis that takes an input program P with a program point C or a collection of C and provides a collection of program points S where either C influences the behavior of P at S or S influences the behavior of P at C. In this concept, C is slice criteria which is a program point or a collection of input program points and S is program slice of P which is also the collection of output program points. In many literatures, the criteria are composed of a number of variables that determine the output or effect of the activities or function. Program slice in which the slice influences the criteria is referred to as backward slices and forward slice is also the program slice in which the criteria influences the slice.

The parametric inter-procedural slicing algorithm (Fig. 1) is a program slicing technique which can calculate the transitive closure of the union of a given set of dependence relations when vertical inter-procedural data flow is handled across invocation sites differently. This algorithm can accept the input program with the slice criteria. Then it considers the set of dependence relations when program slicing is performed. In this algorithm, the initial set of slice criteria is called as seed slice criteria. Starting from this seed slice criteria, the algorithm considers the slice as a fixed point of the function including this criterion and the slice and perhaps empty slice.

Slice (P, C, D, seedCritGenerator, depHandler, procAscHandler, procDscHandler)

1. *workset*: a set of program point and method pairs.
2. *S*: the set of program point in the slice
3.
4. S ← 0
5. *processed* ← 0
6. *workset* ← seedCritGenerator(C)
7. **while** *workset*≠0
8. **do** c ← *remove(workset)*
9. S ← S ∪ {c#1}
10. processed ← processed ∪ {c}
11. **for each** → ∈ D
12. **do** workset ← worksetUdepHandler(c, $\xrightarrow{d}$)\processed
13. **for each** c_n ∈ procAscHandler(c,P) ∪ procDscHandler(c,P)
14. **do if** c_n ∉ processed
15. **then** workset ← workset ∪ {c_n}
16. **return** *S*

Fig 1. A parametric inter-procedural slicing algorithm

According to the parametric inter-procedural slicing algorithm, C is the slice criteria and it is composed of one or more slice criterion c = <e> where *e* is defined as a program point in the input program P. In this program P, every program point is unique. #i indicates the i-th element in the given tuple. The procedure invocation expression of the program point e is *e* ↑ i which means that the program point *e* has the invocation expression of the i-th argument. For dependences <μ,v>, the dependence relation of μ and v denotes (μ$\xrightarrow{d}$v) where $\xrightarrow{d}$ is dependence relation, μ is the source and v is the destination of

the dependence and D is a set of dependence relations, in the case of identifier based data dependence.

Backward-SeedCritGenerator(C,D)
1. return C

Backward-DepHandler(c, $\xrightarrow{d}$)
1. result $\leftarrow$ 0
2. **for each** <μ,c> ∈ $\xrightarrow{d}$
3. **do** result $\leftarrow$ result ∪ {<μ>}
4. **return** result

Backward- ProcAscHandler(c, P)
1. result $\leftarrow$ 0
2. e $\leftarrow$ c #1
3. m $\leftarrow$ OccurringProcedure (e,P)
4. **for each** v ∈ CallSites (m,P)
5. **do for each** i ∈ UseParameters (e,P)
6. **do** result $\leftarrow$ result ∪ {<$\uparrow_i$v> }
7. result $\leftarrow$ result ∪ {<v>}
8. result $\leftarrow$ result ∪ {<$\uparrow_o$v> }
9. **return** result

Backward-procDscHandler(c,P)
1. result $\leftarrow$ 0
2. v $\leftarrow$ c#1
3. **If** ContainsCallSites(v,P)
4. **then for each** m ∈ Callees(v,P)
5. **do for** r ∈ ExitPoints(m,P)
6. **do** result $\leftarrow$ result ∪ {<r> }
7. **return** result

Fig 2. Backward slice generating parameters for the inter-procedural slicing algorithm

In the above figure, backward slicing technique is applied in these algorithms: BACKWARD-SEEDCRITGENERATOR, BACKWARD-DEPHANDLER, BACKWARD-PROASCHANDLER, and BACKWARD-PRODESCHANDLER. In PROASCHANDLER, *OccurringProcedure* returns the procedure in which the given program point occurs in the program and *UsedParameters* returns the index of the parameters used in the given program point in the given program. The *CallSites* returns program point of the call sites where the given procedure is invoked. In the PRODESCHANDLER, *ContainsCallSite* returns true if a call site is included in the given program point and *Callees* returns the set of procedures which are executed at the call site in the given program point in the given program. *ExitPoints* returns the set of exit program points [3].

IV. SYSTEM DESIGN

The design of the proposed system is presented in the following Fig. 3 and it has three parts: converting input C program into java, slicing with parametric inter-procedural slicing to obtain sliced program and measuring code coverage to obtain the improved code coverage. In the first part of the system, the input C program is converted into java and then it is generated into intermediate representation which is also called Jample representation or byte code including the criteria, in the second part of the system. In program slicing, parametric inter-procedural slicing algorithm performs in backward direction (i.e., it starts at the criteria and performs backward in direction until it reach at the end of the program). Finally, the sliced program is obtained by using parametric inter-procedural slicing algorithm. In the third part of the system, the output sliced program is measured by using code coverage metric. Finally, it is ensured that the code coverage of sliced program is more improved than that of the original input program.

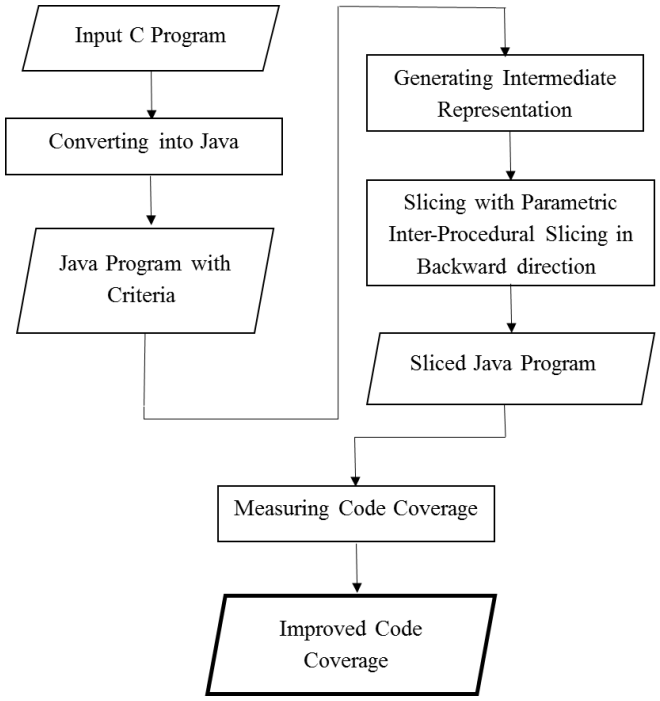

Fig.3 System Design

A. Motiving Example

In this example, the steps of the performance of program slicing on a small program are described. The original input program code is shown in Fig. 4. According to the parametric inter-procedural slicing algorithm, criteria are the statements which can produce the output in the main function. In this figure, criteria C is line 10 and thus, z is output.

```
1.  if(c==true)
2.   flag=1
3.    else  flag=0
4.  y=2
5.  if(d==true)
6.  x=4
7.  else x=5
8.  if(flag==1)
9.  z=x+1
10. else z=y+1

Criteria (C): z=y+1
```

Fig. 4 Example Program Code

By applying parametric inter-procedural slicing algorithm, the backward slicing technique starts at the criteria and then considers its dependencies (10-8-4-3-2-1) in backward direction. Therefore, the desired path (Path No.3) is obtained and Path No. 1 and 2 are not required in execution as shown in Table I.

TABLE I. Possible paths for the output criteria in the example program code

No.	Path	Branch Condition

1	1-2-3-4-5-6-8-10	if(c==true) False
		if(d==true) True (not required in execution)
		if(flag==1) False

2	1-2-3-4-5-7-8-10	**if**(c==true) False **if**(d==true) False (not required in execution) **if**(flag==1) False
3	1-2-3-4-8-10	**if**(c==true) False **if**(flag==1) False

Therefore, the output sliced program is obtained as shown in Fig. 5.

```
1. if(c==true)
2.    flag=1;
3. else flag=0;
4. y=2;
5. if(flag==1)
6. else z=y+1;
```

Fig.5 Output Sliced Program

After program slicing, the output program needs to be measured for code coverage by using coverage metric as shown in (1).

$$cov = \frac{br_{exec}}{br_{total}} \times 100\% \quad (1)$$

In this equation, brexec is the number of program code exercised in a program, and brtotal is the total number of program code in a program. By applying (1), the code coverage of original input program is 50% before slicing and the code coverage of sliced program is 67% after slicing. Therefore, parametric inter-procedural slicing can improve the code coverage of the program source code.

Moreover, cyclomatic complexity of the source code can be compared by using (2),

$$V(G) = P + 1 \quad (2)$$

V(G) is the cyclomatic complexity value of the graph G and P is the number of predicate nodes contained in the graph G. The graphs Gs of original input program and sliced program are as presented in the following Fig. 6.

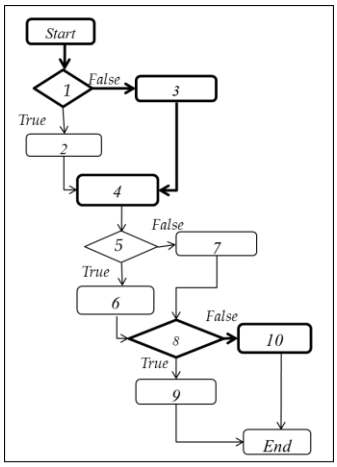

Fig.6 (a) The Graph G of original input program

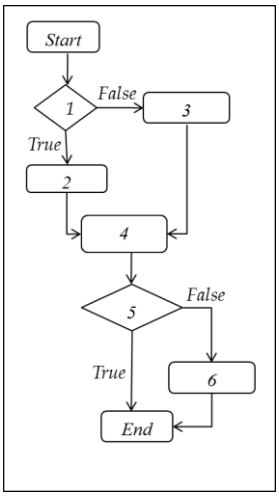

Fig.6 (b) The Graph G of output sliced program

By applying (2), the cyclomatic complexity value of original input program (Fig. 6(a)) is 4 and the cyclomatic complexity value of sliced program (Fig. 6(b)) is 3. By comparing the cyclomatic complexity of Graph G of Fig. 6(a) and Fig. 6(b) before aFTnd after slicing, the cyclomatic complexity of Fig. 6(b) is less than the cyclomatic complexity of Fig. 6(a). Therefore, the complexity of the source code is reduced effectively.

V. Experimental Result

This paper is implemented for improving code coverage for structural testing by using parametric inter-procedural slicing algorithm. In an experiment, to improve coverage of java program, *ntdrivers-simplified*, *ssh-simplified* and *locks* category of *SV-COMP 2013* benchmark dataset is used. In the category of *ntdrivers-simplified*, there are four programs: *kbfiltr*, *diskperf*, *cdaudio* and *floppy* are included. In the category of *ssh-simplified*, there are two programs: *sshserver* and *sshclient* are included. In the category of *locks*, there are

six programs: *locks-5, locks-6, locks-7, locks-8, locks-9* and *locks-10* are included.

Firstly, these are C programs and they are converted into java by using a tool, C++ to Java converter. And then the corresponding intermediate representation (Jample or three-address representation code) byte code is generated by loading SOOT which is Java Optimization Framework [10]. According to the parametric inter-procedural slicing algorithm, *Indus Kaveri* slicing tool in [12] provides the sliced program with the appropriate criteria [13]. The comparing results of total lines of codes (LOC) and cyclomatic complexity values of the original input program and that of the sliced program are as shown in Table II and III. These values are obtained by visualizing and analyzing the source code with the help of *Sci Tool, Understand*. As shown in Table IV, the final coverage result of original input program and sliced program are improved by using *EclEmma* coverage tool after program slicing.

TABLE II. Comparing Total Lines of Code in original program and sliced program

Category of dataset	Total lines of code in Original Program	Total lines of code in Sliced Program
kbfiltr	752	21
diskperf	1079	142
cdaudio	2477	259
floppy	1814	174
sshSever	728	222
sshClient	638	22
Locks-5	128	12
Locks-6	149	25
Locks-7	170	25
Locks-8	191	25
Locks-9	212	26
Locks-10	233	15

TABLE III. Comparing Cyclomatic Complexity values of original program and sliced program

Category of dataset	Cyclomatic ComplexityValue of Original Program	Cyclomatic ComplexityValue of Sliced Program
kbfiltr	31	4
diskperf	25	10
cdaudio	53	10
floppy	35	15
sshSever	101	37
sshClient	90	2
Locks-5	17	2
Locks-6	20	4
Locks-7	23	4
Locks-8	26	4
Locks-9	29	4
Locks-10	32	2

After program slicing, the total lines of codes and cyclomatic complexity in sliced program is more reduced than that of the original as shown in Table II and III. Then, measuring code coverage by using coverage metric, the code coverage of sliced program is more improved than that of the original as described in Table IV. Thus, parametric inter-procedural slicing can improve code coverage for structural testing.

TABLE IV. Comparing Code Coverage of original program and sliced program

Category of dataset	Branch Coverage of Original Program	Branch Coverage of Sliced Program
kbfiltr	33.2%	56.5%
diskperf	16.8%	29.9%
cdaudio	12.6%	43%
floppy	14.8%	51.2%
sshSever	28%	28%
sshClient	33.1%	85%
Locks-5	52.7%	62.5%
Locks-6	52.3%	52.4%
Locks-7	52%	52.4%
Locks-8	51.8%	52.4%
Locks-9	51.6%	52.4%
Locks-10	51.4%	58.3%

Most of the coverage result of sliced java programs are more increased than that of the original programs but the result of only some programs (Locks-6 and Locks-7) are similar to the result of sliced program and the result of *sshServer* remains unchanged. In other programs, improving coverage of sliced programs are described obviously in Table IV. Although the code coverage of *sshClient* is only 33.1% before slicing, the code coverage of this program is 85% after slicing. Thus, the code coverage of sliced programs are more improved than that of the original effectively.

VI. CONCLUSION

The structural testing is high complexity technique and it has low coverage values. Therefore, this paper focuses on program slicing for more effective in improving performance

of measuring code coverage. The parametric inter-procedural slicing is appropriate for program slicing technique because it performs starting at the criteria and considers all dependencies in the given program. By comparing the original program and sliced program before and after applying parametric inter-procedural slicing algorithm, this proposed system ensures that the coverage performance is more improved and the complexity is reduced significantly. Moreover, the tested dataset is benchmark dataset which is converted into java for this experiment.

REFERENCES

[1] Arora, J, "Static Program Slicing- An Efficient Approach for Prioritization of Test Cases for Regression Testing", International Journal of Computer Applications (0975 – 8887), vol. 135 – No.13, pp. 129-132, Feb. 2016.

[2] M. Weiser, "PROGRAM SLICING," presented at the ICSE'81, University of Maryland, 1981, pp. 439–449.

[3] V. P. Ranganath, "Scalable and accurate approaches for program dependence analysis, slicing, and verification of concurrent object oriented programs," 2006, Accessed: 17-Mar-2020. [Online]. Available: https://krex.k-state.edu/dspace/handle/2097/248.

[4] A. Sakti, G. Pesant, and Y.-G. Gueheneuc, "Instance generator and problem representation to improve object oriented code coverage," IEEE Transaction on Software Engineering, vol. 41, no. 3, pp. 294–313, Mar. 2015, doi: 10.1109/TSE.2014.2363479.

[5] E. Alatawi, T. Miller, and H. Sondergaard, "Using metamorphic testing to improve dynamic symbolic execution," in 2015 24th Australasian Software Engineering Conference, Adelaide, SA, Australia, Sep. 2015, pp. 38–47, doi: 10.1109/ASWEC.2015.16.

[6] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan, "DASE: document-assisted symbolic execution for improving automated software testing," in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, May 2015, pp. 620–631, doi: 10.1109/ICSE.2015.78.

[7] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), Graz, Austria, Apr. 2015, pp. 1–12, doi: 10.1109/ICST.2015.7102580.

[8] K. Satyan, "Enhancement of branch coverage using java program code transformer", National Institute of Technology Rourkela, Orissa, India, 2015.

[9] V. Sangeetha, T. Ramasundaram, "Optimizing whole test suite generation", International Journal of Advanced Research in Computer and Communication Engineering Vol. 5, Issue 1, 2016.

[10] V. P. Ranganath and J. Hatcliff, "Slicing concurrent java programs using indus and kaveri," International Journal on Software Tools for Technology Transfer, Vol. 9, pp. 489-504, 25 Jul. 2007.

[11] H. Watson, J. McCabe, "Structured testing: a testing methodology using the cyclomatic complexity metric", NIST Special Publication 500-235, 1996.

[12] G. Jayaraman, V. P. Ranganath, and J. Hatcliff, "Kaveri: delivering the indus java program slicer to eclipse," Proceedings of the 8th international conference, joint European Conference on Theory and Practice of Software conference on Fundamental Approaches to Software Engineering, Mar. 2005, Lecture Notes in Computer Science 3442, pp. 269-272, doi: 10.1007/978-3-540-31984-9_20.

[13] S. Bardin, N. Kosmatov, and M. Delahaye, "Enhancing symbolic execution for coverage-oriented testing⋆," CEA LIST, Software Safety Lab, Paris-Saclay, France, 2015.

[14] Y. Dubey, D. Singh, and A. Singh, "Amalgamation of automated test case generation techniques with data mining techniques: A survey," International Journal of Computer Application (IJCA), vol. 134, no. 5, pp. 18–22, Jan. 2016, doi: 10.5120/ijca2016907950.

[15] M. M. Aung and K. T. Win, "Reducing Complexity of Java Source Codes in Structural Testing by Using Program Slicing", IJC, vol. 30, no. 1, pp. 78-85, Aug. 2018.

[16] M. M. Aung and K. T. Win, "Improving Branch Coverage for White-box Testing," 27th International Conference on Computer Theory and Applications (ICCTA), Alexandria, Egypt, Oct, 2017, pp. 63-68.