International Journal of Computer (IJC)

ISSN 2307-4523 (Print & Online)

http://ijcjournal.org/

Reducing Complexity of Java Source Codes in Structural Testing by Using Program Slicing

Myint Myitzu Aung[a]*, Kay Thi Win[b]

[a,b]*University of Computer Studies, Mandalay, Myanmar*

[a]*Email: myintmyitzuaung@gmail.com,* [b]*Email: kthiwin11@gmail.com*

Abstract

Structural testing is one of the techniques of software testing. It tests only the structure of the source code while comparing expected results and actual results. Generally, structural testing takes a long time to perform its task and not possible. Sometimes, only a small portion of the program is relevant. This can be done by program slicing. Program Slicing is to decompose the program into smaller units that depends on different types of dependencies between the program statements. The different types of program slicing are forward slicing, backward slicing, complete slicing, dynamic and static slicing, etc. Moreover, there is Tree Slicing which is also a key technique to slice and merge different Symbolic Execution (SE) sub-trees under some specific conditions. In this paper, we combine Tree Slicing technique and Indus Kaveri where Indus is a robust framework for analyzing and slicing concurrent Java programs, and Kaveri is a feature-rich Eclipse-based GUI front end for Indus slicing. Then we present the experimental results in order to reduce the complexity of the java source code.

Keywords: Program Slicing; Tree Slicing; Symbolic Execution.

1. Introduction

In the software development process, software testing plays an import role. The software testing can compare the expected and actual result of software by executing a program with the purpose of finding different types of faults. There are two types of software testing one is functional testing and another one is structural testing. In the case of functional testing, it is based on the functional part of the system and ignores internal details while comparing actual and estimated result. In the case of structural testing, it is focused on internal program structure while comparing expected and actual result and finding out faults. Therefore, structural testing is the process of evaluating a system by comparing its actual and expected result manually or automatically.

---

* Corresponding author.

But structural testing takes a long time to perform completely and not possible. Sometimes, for many properties, only a small portion of the program is relevant. This can be done with the help of slicing. Slicing is an important testing technique, it helps in understanding of the program or software by decompose the program into smaller parts depending on the different types of dependencies (data, method call, control, etc) between the statements. In program slicing, each slice only containing statement that relevant to specific variable and ignore other statements. There are many types of program slicing approaches depending upon the run-time environment and slicing direction. Depending upon the run time environment, slicing can be static or dynamic and depending upon the slicing direction, slicing can be forward or backward slicing [1].

In this paper, one of the program slicing techniques, Tree slicing is used. Then, we combine it with Indus Kaveri where Indus is a robust framework for analyzing and slicing concurrent Java programs, and Kaveri is a feature-rich Eclipse-based GUI front end for Indus slicing. Tree Slicing is also called as Path Sensitively Sliced Control Flow Graph (PSS-CFG) which is a key technique to slice and merge different Symbolic Execution (SE) sub-trees under some specific conditions. The background theories are shown in section 3 and analysis of its results are presented in section 4.

2. Related Work

A Tamrawi, S Kothari introduced the notion of event-flow graph (EFG) and presented a lineartime algorithm to calculate equivalence classes by compacting a Control Flow Graph (CFG) into an EFG. Each path in the EFG represents an equivalence class of paths in the CFG. They showed that it is enough to perform path-sensitive analyses only on the equivalence classes produced by an EFG rather than on all the individual paths in the CFG [2].

J Jaffar and his colleagues presented a fully path-sensitive backward slicer limited only by solving capabilities and loop invariant technology. The major result is a symbolic execution algorithm which avoids ambiguity due to infeasible paths and joins at merge points and halts execution of a path if certain conditions hold while reusing dependencies from already executed paths. The conditions are focused on an idea of interpolation and witness paths to detect “a priori” whether the exploration of a path could increase the accuracy of the dependencies computed so far by other paths. They demonstrated the experiment of this approach with real medium-size C programs [3].

C Hammer and his colleagues presented a system for information flow control in Java programs and it is based on path conditions in dependence graphs. Such path conditions are very precise necessary conditions for information flow between two program points. Their approach is fully automatic, flow-sensitive, context-sensitive, and object-sensitive. Their results indicate that the number of false alarms is drastically reduced compared to type-based IFC systems, while of course all potential security leaks are discovered [4].

G Jayaraman and his colleagues described a system which is a modular program slicer for Java built using the Indus program analysis framework along with its Eclipse-based user interface called Kaveri. Indus provides a library of classes that enables users to quickly assemble a highly customized non-system dependence graph

based inter-procedural program slicer capable of slicing concurrent Java programs. Kaveri is an Eclipse plugin that relies on the above library to deliver program slicing to the eclipse platform. In this paper, the authors described that apart from the basic feature for generating program slices from within eclipse along with an intuitive UI to view the slice, the plugin also provides the capability for chasing various dependences in the application to understand the slice [5].

3. Background

3.1. Path-Sensitively Sliced Control Flow Graph (PSS-CFG)

It is Tree Slicing, a key technique to slice and merge different Symbolic Execution (SE) sub-trees under some specific conditions. To obtain the transformed program, two-steps algorithm is used. First step is to generate SETree annotated with dependencies. Second step is to transform the tree by removing sub-tree and edges, to obtain the PSS-CFG. To generate SETree annotated with dependencies, the following three transformation rules and algorithms are used [6].

- Rule 1: The statement can be removed if the LHS of an assignment statement does not include in the dependency set.
- Rule 2: If a branch point has only one feasible path which arises from it, it can be replaced or removed.
- Rule 3(called “Tree Slicing”): If both the “then” and “else” cases include no statement which is included in the slice, an entire branch is inappropriate to the target point and can be removed.

GENPSSCFG $(v \equiv \langle \ell, s, \Pi \rangle)$
1: **if** $\exists\, v' \equiv \langle \ell, s', \Pi' \rangle$ s.t. v and v' satisfy Eqn. 4
2: **then** MERGE (v, v')
3: **else if** v is at a branch point **then**
4: SPLIT(v)
5: **else**
6: SYMEXEC (v)

Figure 1: Generating PSS-CFG

MERGE (v, v')
1: $\overline{\Psi}_v := \overline{\Psi}_{v'}$
2: $\sigma_v := \sigma_{v'}$
3: $\mathcal{S} := \mathcal{S} \cup \mathsf{merged}(v, v')$

Figure 2: Merging

SPLIT $(v \equiv \langle \ell, s, \Pi \rangle)$
1: $\overline{\Psi}_v$:= true
2: **foreach** transition $\ell \xrightarrow{\text{assume(c)}} \ell'$ **do**
3: **if** (v is a loop header) **then**
4: $v' \triangleq \langle \ell', \cdot, \text{invariant}(v) \wedge [\![\text{c}]\!]s \rangle$
5: **else**
6: $v' \triangleq \langle \ell', s, \Pi \wedge [\![\text{c}]\!]s \rangle$
7: **if** v' is infeasible state **then**
8: $\mathcal{S}$:=$\mathcal{S} \cup \text{inf\_edge}(v \xrightarrow{\text{assume(c)}} v')$
9: $\overline{\Psi}_{v'}$:= false, $\sigma_{v'} := \emptyset$
10: **else**
11: $\mathcal{S}$:=$\mathcal{S} \cup \text{edge}(v \xrightarrow{\text{assume(c)}} v')$
12: GENPSSCFG (v')
13: $\overline{\Psi}_v := \overline{\Psi}_v \wedge \widehat{wlp}\,(\overline{\Psi}_{v'}, \text{assume(c)})$
14: $\sigma_v := \sigma_v \sqcup \widehat{pre}\,(\sigma_{v'}, \text{assume(c)}, s)$
15: **if** $\delta \equiv v \xrightarrow{\text{assume(c)}} v'$ satisfies Eqn. 3 **then**
16: $\mathcal{S}$:=$\mathcal{S} \cup \text{in\_slice}(v \xrightarrow{\text{assume(c)}} v')$

Figure 3: Splitting

SYMEXEC $(v \equiv \langle \ell, s, \Pi \rangle)$
1: **if** $\nexists$ transition relation $\ell \xrightarrow{\text{x:=e}} \ell'$ **then**
2: $\overline{\Psi}_v$:=true, $\sigma_v := \mathcal{V}$
3: **else**
4: $v' \triangleq \langle \ell', s[x \mapsto [\![e]\!]s], \Pi \rangle$
5: $\mathcal{S}$:=$\mathcal{S} \cup \text{edge}(v \xrightarrow{\text{x:=e}} v')$
6: **if** v' is not a loop header
7: GENPSSCFG (v')
8: $\overline{\Psi}_v := \widehat{wlp}\,(\overline{\Psi}_{v'}, \text{x:=e})$
9: $\sigma_v := \widehat{pre}\,(\sigma_{v'}, \text{x:=e})$
10: **if** $v \xrightarrow{\text{x:=e}} v'$ satisfies Eqn. 2 **then**
11: $\mathcal{S}$:=$\mathcal{S} \cup \text{in\_slice}(v \xrightarrow{\text{x:=e}} v')$

Figure 4: Symbolic Execution

3.2. Correctness of Transformation Theorem

Theorem: *By applying RULE 1, RULE 2 or RULE 3 to a CFG(G), a transformed CFG(G0) in which G0 is equivalent to G with respect to the target variables V.*

The proof of the correctness of Tree Slicing can be performed as follows. Assume that there is a path in G starting from Vstart to V0 and then reaches V1. Assume that the condition c1 holds at V1, so it follows V2, reaches the merged point Vk and then continues to reach the terminal, Vend. Let us call this path πG. In G' which is obtained by using Tree Slicing on G, thereby removing the entire branch at V1, the same input may follow a path, say πG', exactly, πG' looks like a path starting from Vstart till V0 in πG , therefore V0 is the same symbolic state. At this point, πG' is different from πG by implicitly "skipping" the execution at V_1 and instead directly reaches V_k. Since V_k and V'_k are merged, the dependency sets are the same at both points. Now,

since the transition from V_1 to V_2 with condition c_1 in G was not included in the slice. This implies that the symbolic state of the path $\pi G'$ at V_k is the same as the symbolic state of the path πG at V_k as far as the dependency variables at V_k are concerned. Exactly, the values of the dependency variables at V_k are the same in both πG and $\pi G'$. Since these are the only variables affecting the target variables V at V_{end}, it is ensure that $\pi G'$ will generate the same values for V as πG. Of course $\pi G'$ may generate different values than πG for variables not in V. Until fixed point is reached, three rules are applied repeatedly (i.e., these cannot be applied anymore). Soundness of individual rule applications is guaranteed from this Theorem. Transitiveness of the rules is also guaranteed by Theorem because each new CFG is equivalent to the original CFG. Thus, this Theorem guarantees that the PSS-CFG is equivalent to the original program with respect to the target variables V [6].

3.3. Indus Java Program Slicing Framework

The primary features of the architecture of the Indus slicer are

- Intermediate Representation: java programs are represented in Jample, a type of three address representation provided by SOOT,
- Batteries Included: various dependence analysis, and analyses to calculate and prune various dependences – intra- and interprocedural data dependence, control dependence, interference dependence, ready dependence and so on are included,
- Loose Coupling, Modularity: - analyses are available as independent modules,
- Customizability: the user can choose the residualization by clone or update.

Moreover, the advance features are also included. They are :

- Non-SDG based Dependence/Slicing: slicing based on system dependence graphs, dependence information is reusable, fine-tuning of slicing algorithm is simplified, and maintenance becomes easy,
- Program Slicing: is Program Analysis,
- Calling Context Sensitive Slicing: slicing algorithms that support calling context insensitive and support by keeping track of calling contexts while descending into call sites and tracing back the recorded calling contexts are said to be calling context sensitive. Indus supports both calling context insensitive and calling context sensitive slicing of sequential programs,
- Context-restricted Slicing: is useful in debugging applications based on an exception stack trace, i.e., a user would like to calculate the slice that affects only the parts of the program occurring on an exception stack trace,
- Scoped Slicing: is useful for removing parts of the runtime libraries and helps in checking for data confinement in the realm of security,
- Concurrent Java Program Slicing: leverages the escape analysis to rule out cases,
- Complete Slicing, Chopping, and Control Slicing: a slice that contains parts of the program that affect and are affected by the slice criteria and every program point in the slice [7].

4. Evaluation and Analysis

In experiments, the device drivers from the *ntdrivers-simplified of SV-COMP 2013* benchmark dataset is used. This dataset is C programs and these programs are converted into java by using *C++ to java converter*. These converted java programs are sliced by using two steps algorithms. In *in_slice* step of *Splitting* (Fig. 3), Indus Kaveri tool is applied. In the residualization of this Indus, the appropriate rule of three rules is applied. After slicing the program, the original program is reduced by removing unused statements, method and inappropriate code with specific criteria. The comparing of original and transformed program is as shown in Table 1.

Table 1: Comparing total lines of code, methods and statements included in the original and transformed source codes

	Original Source Code				**Transformed Source Code**			
	kbfiltr	diskperf	ssh Server	ssh Client	kbfiltr	diskperf	ssh Server	ssh Client
Total Lines of Code	584	1079	728	638	20	111	101	22
Total number of Methods	31	54	47	40	1	7	12	3
Total number of Statements	344	662	435	388	14	58	43	12

To express the complexity of source code, cyclomatic complexity numbers are needed to compare. There are ten complexity metrics are used in comparing the complexities of original and sliced transformed programs.

These metrics are

(1) Cyclomatic complexity,

(2) Essential complexity,

(3) Maximum cyclomatic complexity,

(4) Maximum modified cyclomatic complexity,

(5) Maximum strict cyclomatic complexity,

(6) Maximum essential complexity,

(7) Sum of cyclomatic complexity,

(8) Sum of modified cyclomatic complexity,

(9) Sum of strict cyclomatic complexity,

(10) Sum of essential complexity.

In order to the purpose of this paper, the complexity values are decreased clearly by using the technique of program slicing. These complexity values are collected as shown in Table 2 by using code visualizer *SCiTool, Understand.*

Table 2: Comparing Cyclomatic Complexity of four categories of benchmark dataset

Category of dataset	**Cyclomatic**	**Essential**	**Max Cyclomatic**	**Max Cyclomatic Modified**	**Max Cyclomatic Strict**	**Max Essential**	**Sum Cyclomatic**	**Sum Cyclomatic Modified**	**Sum Cyclomatic Srtict**	**Sum Essential**
Kbfiltr	31	3	31	31	31	3	97	97	97	33
kbfiltr Slice	4	1	4	4	4	1	4	4	4	1
Diskperf	25	10	25	25	25	10	146	146	146	73
diskperf Slice	17	1	17	17	17	1	23	23	23	7
ssh Client	90	1	90	90	90	1	129	129	129	40
ssh Client Slice	2	1	2	2	2	1	4	4	4	3
ssh Server	101	40	101	101	101	40	147	147	147	86
ssh Server Slice	11	1	11	11	11	1	22	22	22	12

5. Conclusion and Recommendations

The necessary for reducing the complexity of structural testing can be completed by using several approach of excluding the infeasible branches and program slicing techniques. This paper proved that reducing the complexity of java source code by using the knowledge of Path Sensitively Sliced Control Flow Graph (PSS-CFG) or Tree slicing with the help of Indus Kaveri. This combination technique for reducing complexity can perform depending on the specific criteria and it removes inappropriate branches of this criteria, unused statements and blanks of codes. Therefore, it can reduce the complexity of java source code in structural testing.

Reducing the complexity of java source code can improve the java code coverage such as path coverage, decision\condition coverage, statement coverage, loop coverage and so on. As a limitation, we applied only four categories of benchmark dataset by using this technique which is ensured to reduce the complexity of these java source codes.

References

[1] J Arora, "Static Program Slicing- An Efficient Approach for Prioritization of Test Cases for Regression Testing", International Journal of Computer Applications (0975 – 8887) Volume 135 – No.13 (2016), 18-23

[2] A Tamrawi, S Kothari, "Event-Flow Graphs for Efficient Path-Sensitive Analyses", arXiv preprint arXiv:1404.1279, 2014 - arxiv.org

[3] J Jaffar, V Murali, J A. Navas, and A E. Santosa, "Path-Sensitive Backward Slicing, International Static Analysis …", 2012 – Springer

[4] C Hammer, J Krinke, G Snelting, "Information Flow Control for Java Based on Path Conditions in Dependence Graphs", Proceedings IEEE International Symposium on Secure Software Engineering, 2006.

[5] G Jayaraman, V Prasad Ranganath, and J Hatcliff, "Kaveri: Delivering the Indus Java Program Slicer to Eclipse", DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by NSF (CCR-0306607) by Lockheed Martin, and by Intel Corporation.

[6] J. Jaffar, V. Murali, "A Path-Sensitively Sliced Control Flow Graph", presented at FSE '14, November 16-22, 2014, San Hong Kong, China. Copyright 2014 ACM.

[7] V Prasad Ranganath · J Hatcliff, "Slicing Concurrent Java Programs using Indus and Kaveri", in International Journal on Software Tools for Technology Transfer, 1-15