High Performance Computation Through Slicing and Value Replacement with CCDD Approach

Deepti Tak, Shalini Rajawat, Vijay Singh Rathore

Abstract: In software development and maintenance stages, programmers need to frequently debug the software. Software fault localization is one of the most exclusive, tedious and time intense activities in program debugging. A common approach to fix software error is computing suspiciousness of program elements according to failed test executions and passed test executions. However, this technique does not give full consideration to dependences between program elements and therefore it reduce the ability for efficient fault localization. Developers must identify statements involved in failures and select suspicious statements that may contain faults. Our paper presents a new technique that identify statements involved in failure –those executed by failed test cases through narrowing the search domain using Slicing Technique (Control and Data dependence slice) by slicing the program and making it more effective with the CCDD (Coupling Control and Data Dependency) approach in Value Replacement. The proposed approach is more efficient and is more accurate in locating statements that directly\indirectly effect the faulty statements. This approach can also be applied to many other research areas.

Keywords: CCDD (Coupling Control and Data Dependency) approach, Slicing Technique, Value Replacement

I. INTRODUCTION

Debugging bugs in software program is expected to consume 50% to 80% of the development and maintenance effort **[1].** Clearly, techniques that can reduce the time required to locate faults can have a significant impact on the cost and quality of software development and maintenance.
Pan and Spafford after analyzing and observing the debugging process consistently, concluded four tasks is required in locating the fault in a program are:

(A) Identify statements involved in failures-those executed by failed test cases;
(B) Narrow the search domain as much as possible by selecting suspicious statements that might contain errors;
(C) Theorize about suspicious faults by techniques.
(D) Restore program variables to a specific state .

Our proposed work focuses mainly on the second task-selecting suspicious statements that may contain the fault and reduce the search domain by using slicing and value replacement technique with our CCDD(Coupling Control and Data dependency) approach.

Revised Manuscript Received on February 06, 2013.

Deepti Tak, Jagannath Gupta Institute of Engineering & Technology, Jaipur, India.
Dr. Shalini Rajawat, Vivekananda Institute of Technology, Jaipur, India.
Dr. Vijay Singh Rathore, Karni College, Vaishali Nagar, Jaipur, India.

We propose a new approach CCDD for fault localization which (1) Use slicing (static control and Dynamic Data information) to narrow the search domain or statements ,then (2) Calculate suspiciousness of fault by predicting statement set CCDD Set(Data And Control information) using Value replacement .The proposed approach concludes that adding control dependence with data dependence is more valuable in suspicious analysis if compared to any other previous [2,3,4,5,6] techniques known so far. In the end, we get the set of the statements which are greater likelihood of being faulty.

II. BACKGROUND AND OUR APPROACH

A. Program slicing

Program slicing [7-11] is a program analysis technique which is used for slicing the big program into tiny and plain code fragments. It is generally used for program analysis, testing, debugging, understanding, metric, etc. In 1979, Mark Weiser [7] first proposed concept of program slicing, where he defined program slice of program p is an executable part of the p, in terms of variable s of interest point v, the executable part of program is equal to the program p in function .Or In more appropriate words ,we say in a given a source program ***p,*** *program slice* is a group of statements probably affecting the value of *slicing criterion* (a pair **<s, *v*>, s** is a statement in ***p*** and **v** is a variable defined or referred at **s).**

We classify Slicing as static slicing and dynamic slicing. The main difference between them is the information they required. Static program slicing only needs static information which is the source code, while Dynamic slicing requires the entire execution trace which is resultant to a precise program input. In additional, static slicing considers all the potential executions where as dynamic slicing only considers the execution of a definite input. As a result, dynamic slicing include smaller number of statements and is more accurate than the static one. With complex program and long execution traces, dynamic slicing is not very practical.

Slice calculation is based on dependence analysis between program statement in a source program, where dependence analysis has two components, data dependence analysis and control dependence analysis. In existing software development environments, Object- Oriented languages like JAVA and C++ have new concepts such as *class*, *inheritance*, *dynamic binding* and *polymorphism* .Since Object-Oriented languages have many dynamically determined elements, static slicing cannot figure out practical (or precise) analysis results. Alternatively, dynamic slicing needs to sketch execution trace, it involve too much time and memory space.

B. How to Compute Program Slice?

Now, we will in brief explain the process of computing a program slice as follows.

STEP 1: We will first identify defined variables and pass on ones for each statement in a source program.

STEP 2: Secondly, We will extract data and control dependence relations between program statements.

STEP 3: Thirdly,We construct *Program Dependence Graph (PDG)* Using dependence **(Control and Data dependence slice)** relations extracted on step 2.

STEP 4: Lastly**,** We compute the slice for the slicing criterion specified by the user.

With the intention of computing the slice designed for a slicing criterion **<s,** *v*>, PDG nodes are traverse in reverse order from *Vs,* (node *Vs,* denotes statement **s).**

How to Compute Program* Dependence Relation *?

a. Control dependence (CD)

We take statements **s1** and **s2** in a source program ***p.*** For a *control dependence* ***(CD)*** from statement **s1** to statement ***s2*** all the following condition must be true:

1. Statement **s1** should be a conditional predicate, and

2. The outcome of **s1** determines whether **s2** is executed or not.

Represented by *CD(s1, s***2**) or **s1 --→s2.**

b. Data dependence (DD)

We take statements **s1** and **s2** in a source program ***p.*** For a *Data dependence* ***(CD)*** from statement **s1** to statement ***s2 by a variable v exists if*** all the following condition must be true:

1. Statement **s1** defines *v*,
2. Statement ***s2*** refers v , and
3. **And there a**tleast one execution path exists from **s1** to **s2** without redefining v(ie. *reachable).*

Represented by *DD(s1, v , s 2)* or **SI v--→*s2.***

How to Compute Program Dependence Graph (PDG)?

A PDG is a directed graph whose nodes stand for statements in a source program, and edges represent dependence relations (DD or CD) among statements. **A** DD edge is labeled with a variable name *"a"* if it denotes ***DD (---,****a*,---). An edge from node *Vx,* to node *Vy* represents that "node Vx depends on node *Vy".*

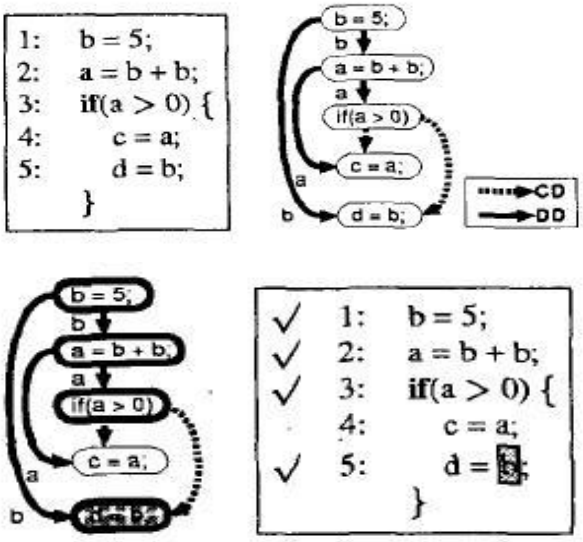

Fig1 and Fig2

Figure 1 shows a sample C program and its PDG **(step 1 - 3),** and Figure 2 shows the slice ("√"-marked statements) for <5, **b**> on Figure 1 **(step 4).**

C. Value Replacement

An automated, dynamic state alteration technique called Value replacement is developed for locating software errors. This technique analyzes program executions that fail due to incorrect output being produced. In such a failing execution, Value substitution alters the execution state at a single statement instance, one after the other, by replacing the set of values involved at that statement instance with an alternate (different) set of values. Execution then proceeds from that point under the altered state. At the end of execution, the output is examined to determine whether or not it has changed to become correct. If the output has become correct, then there is a chance that the statement instance, at which the Value replacement was performed, is faulty. The Value replacement technique performs these value substitution at different statement instances in a failing execution, one at a time, to rank program statements according to how likely they are to be faulty. This is the essence of the Value substitution technique. When value replacements cause the execution of a failing run to become exact, this information is represented by an interesting value mapping pair (IVMP).

What is an Interesting Value Mapping Pair ?

An interesting value mapping pair (IVMP) is a pair of value mappings ("original", "alternate") associated with a particular statement instance in a failing run, such that: (1)"original" is the original set of values used by the failing run at that instance; and (2)"alternate" is an alternate (different) set of values such that if the values in "original" are replaced by the values in alternate" at that instance during execution of the failing run ,then the incorrect output of the failing run becomes correct.

It has been observed that IVMPs often occur at faulty statements or statements that are directly associated to faulty statements through a dependence edge. A dependence sequence with IVMPs in one flaw run might not exist in another flaw run which may have very dissimilar dependence chains. In addition, IVMPs that turn out to recompense for a error in one flaw run are unlikely to recompense for the error in the same way in another flaw run. Taking into consideration runs exercise unusual paths in the program. Hence, we rank IVMP statements keeping in mind that statements coupled with IVMPs in more failing runs are more prone to be faulty than statements that are coupled with IVMPs in smaller number failing runs.

Observations and inferences in value replacement

In support of a incorrect result, we discover that the test case executes a wrong statement in the program. But, Can we be sure that there is no incorrect statements in the program for a correct output. The answer lies in the possibility that the incorrect statements cannot be executed or maybe executed but has no effect on the yield.

For that reason, conclusion of J.Sun are[12]:

1) Any statement not executed under a test case cannot influence the program yield for that test case.

2) And if a statement is executed under a test case it is not necessary that output is affected by the statement.

III. OUR APPROACH IN SLICING & VALUE REPLACEMENT

At current, Object- Oriented programming languages maintain exception-handling mechanism which is used to encourage the robust software program. Exception is an unexpected state or fault occurs in the process of execution at run time. Exception can be classified as *application exceptions* and *runtime exceptions*. The *application exceptions* are explicitly thrown and caught but latter, and *runtime exceptions* are not mandatory to be caught. But if a runtime exception is raised in the program execution is not handled, the program will be terminated. Our CCDD approach uses stack trace to find out the resource statements which cause an exception and presents a new static program slicing algorithm by means of easy dynamic(control information) stack trace. The control information of stack trace is used to conclude the execution path of the program .Moreover stack trace is much lesser than the execution trace (as not including non executed statements), the access and storage of it do not require any extra cost. The correctness of the slice is improved by not including those statements that not linked with this execution. Consequently, our approach can greatly reduce the cost, time and the search domain.

This approach is intended to enhance the precision of static program slicing by using stack traces (**Coupling Control and Data Dependency**). It can also be used to explore the program based on object oriented with stack trace, by specific a function call context. The size of the slice is reduced by some low% than the general approach but surely without losing the accuracy of locating the cause of the exception.

As we all knew during slicing, there is no need to make program dependence graphs equivalent to the non- executed methods (reasons are like exception, function call etc ie. control dependency).

Dependencies information must be calculated for only execute methods and statements, but there are certain statements which are not executed at all should be ignored when performing slicing.

Resultant will be simplified SDG and reduced search domain. When constructing the simplified SDG, there isn't any edge from the methods which are not-executed and therefore no need to handle those methods in performing program slicing.

IV. OUR APPROACH

Modified Observations and inferences in value replacement

Whenever there is an incorrect output in the test case ,we conclude that fault statement is executed. This fault statement is attached to either the control or data dependence of the resultant statement.

We therefore modified the earlier [12] conclusions.

1) The erroneous statement affects the yield by data dependence chain.

2) The erroneous statement affects the yield by control dependence chain which is responsible for the changed executed path which results in the fault output.

3) The erroneous statement affects the yield of the program by both the data and control dependence.

Dennis's[13] approach is not applicable for a state which cannot find the fault in the statement. Because of ignoring the effect of control dependence user get the correct output with the fault statement in the program. If the fault statement is on the data dependence chain, by only changing the IVMPs in control dependence can produces correct output. In point of fact, the test case change the value of switch control which transfer the execute path and give the correct output.

TEST CASE	T1	T2	T3
INPUT	(1,1,2)	(1,1,-1)	(-1,0,2)
CORRECT OUTPUT	11	5	11
WRONG OUTPUT	8	5	11
RESULT	FAIL	PASS	PASS
REASON	ERROR	DD Data dependence	CD controldependence

```
s1: cin >> a>> b;
s2:cin >> c;
 s3: p = a + b + 1;
 s4: q = z + 2;
s5: if (p <= 3)
s6: c = 3+ q*2 ;
else
s7: c = q + 4 ;
s8: cout << c << endl;
```

Fig 2 with table 1

In the program in Figure 2, we suppose that there is a fault in s3 and the statement should be instead of p = a+b+3. TABLE 1 lists three executions that result in one wrong result and two correct results. For each execution, the table shows test case , the input, the wrong output, the correct output, and the result .For T1, we input (1, 1, 2) to debug, but get the unexpected output c = 8 which does not equals to c = 11. We change the value of z to -1 and get the correct output. It is obvious that earlier approach only get the statements set (s4, s6) which does not include fault statement s3. Example can draw a conclusion that only considering the fault on data dependence will give no good for the fault on control dependence.

V. ANALYSIS AND ALGORITHM FOR OUR SLICING AND VALUE REPLACEMENT WITH CCDD APPROACH

Our Approach is divided in 2 phase

Phase 1: (Slicing Approach)

To identify the statement or methods which are not executed at all.

Input-: A program P, with test case set T which include failing running set F due to runtime fault **Output-:** A program P, with test case set T which include failing running set F but with lower search domain or statements.

Procedure

Step 1: When exception occurs, stack trace is stored.

Step 2: We then, infer program execution path by using stack trace, and identify those methods and statements which are not executed at all.

Step 3: Then ,build the simplified SDG that do not consider those methods, which means not computing their dependencies and not adding corresponding parameter-in, parameter-out and summary edges to these methods.

Step 4: Finally, a more accuracy and useful slice will be yielded by using the improved program slicing algorithm.

Phase 2: (Value Replacement)

Input: A program P, test case set T which include failing running set F but with lower search domain or statements

Output: according test case set F we get the set of suspicious statements

Step 1: Using the CCDD approach
Foreach (failed test case T)
{1.Find the statement set depend on data dependency ,then
2.Find the statement set depend on control dependency
3. Make new set by coupling above two set and name it as CCDD set.
}
Step2: Obtain suspicious statement's order
Foreach (statement s in CCDDset)
{ compute suspiciousness(s) to rank the statement and find out and correct the faulty statement.}

VI. CONCLUSION

How slicing technique will benefit in our approach as follows :
(1) Here stack trace information is used to exclude methods which are not associated to the execution in contrast with the earlier program slicing algorithm which include most methods which have nothing to do with the exception,
(2) In case of a complex program with many loops or recursions, it's difficult to infer the executed path therefore lot of methods get included.
(3) Increase the efficiency of finding fault in the statement by considering both Data and Control Dependency, especially for large programs.

REFERENCES

1. J. S. Collofello and S. N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. Journal of Systems and Software, 9(3):191-195, 1989.
2. H. Agrawal and J. R. Horgan. Dynamic program slicing. Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, pages 246{256, June 1990.
3. B. Korel and J. Laski. Dynamic program slicing. Information Processing Letters, 29(3):155{163, October 1988.
4. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. Tools and Algorithms for the Construction and Analysis of Systems, pages 93{107, April 2005.
5. X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 169{180, June 2006
6. X. Zhang and R. Gupta. Cost effective dynamic program slicing. ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 94{106, June 2004.
7. M. Weiser, "Program slicing," In Proceedings of the 5th International Conference on Software Engineering, San Diego, California, United States, 1981.
8. S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," ACM Trans. Program. Lang. Syst., vol. 12, pp. 26- 60, 1990.
9. H. Agrawal and J. R. Horgan, "Dynamic program slicing," In Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, White Plains, New York, United States, 1990.
10. Z. Chen and B. Xu, "Slicing object-oriented java programs," SIGPLAN Not., vol. 36, pp. 33-40, 2001.
11. B. Xu, J. Qian, X. Zhang, and et al., "A brief survey of program slicing," SIGSOFT Softw. Eng. Notes, vol. 30, pp. 1-36, 2005
12. J.Sun,Zhshu Li, Jianchen Ni,et al. Software Fault Localization Based on Testing Requirement and Program Slice[C]: IEEE Computer Socitey, 2007
13. D.Jeffrey, N.Gupta, fault localization Using Value Replacement[C].ACM July 20-24