Efficient Static Analysis with Path Pruning using Coverage Data

Vipindeep V, Pankaj Jalote
Department of Computer Science and Engineering
Indian Institute of Technology
Kanpur, India - 208016
{vvdeep,jalote}@cse.iitk.ac.in

ABSTRACT

Soundness and completeness are two primary concerns of a static analysis tool for finding defects in software. Exhaustive static analysis of the program through all paths is not always possible, especially for a large software causing incompleteness in the analysis. Also, exhaustive testing of the program to detect all bugs is not possible. In this work, we describe a technique which uses coverage data from testing to remove the tested paths and then statically analyzes the remaining code. This pruning of tested paths allows a static analyzer to perform a more thorough analysis of the reduced code, thereby improving its effectiveness. This work is a step towards integration of static analysis and testing frameworks. The proposed technique is applied with a few static analyzers publicly available. Our experience shows that the approach results in lesser false positives as well as detection of more serious errors which might have gone unnoticed otherwise.

Keywords

Testing, Static analysis, Program checking

1. INTRODUCTION

Reliability of the software system is basic concern during development. The only way to achieve high reliability is by detecting and removing the defects present in the program. In general, the behavior of a program can be known only at runtime. However, some of the inconsistencies and errors prevailing in the program can be directly caught by analyzing the code. These errors or bugs present in the program may cause undesired effects during execution. Analyzing the program can identify these errors which might not have been found by the compiler.

Testing and static analysis are prevalent approaches to identify the bugs in the software. Both of these techniques have contrasting features which can be exploited for mutual benefit and better performance in detecting the bugs which will be the focus of this work. [1]

Defects in program can be identified by testing [8]. Here we have a set of test cases exclusively built to execute some conditions and parts of the code which would let us know the possible defects in the program. To identify a defect, we may need a large set of test cases which involves a lot of effort. So the soundness and completeness of testing to detect the defects depends on the test cases used. Soundness refers to the fact that identified defects are correct without being false positives and completeness indicates that all defects in program are being identified. It is obvious that, by testing, the inferred results are valid for the current execution paths and hence unsoundness and incompleteness persists along the paths uncovered by the test case(s).

On the other hand, in static analysis, we need not execute the program and hence the notions of test case does not exist. The analysis is performed on the source code or some representation of program. The representation may be an abstract syntax tree, a dependence graph, a call graph etc [1]. The results observed from such an analysis will remain valid for all input data. Hence we can have guarantee on the applicability of the inferences made by static analysis. Model checkers, annotation checkers etc are different approaches for static analysis tools. There are many static analysis tools available publicly and commercially to detect bugs in the program [7, 3, 4, 5].

The issues of concern in static analysis tools are soundness, completeness and noise. As the complete analysis of the program through all possible paths may not be possible at all times, we may have to exclude some paths during analysis and this forms the basis for incompleteness. The power of the static analysis and testing to detect defects is not the same and each has its strengths . If we can combine the advantages of both the techniques, we can improve the checking of programs. This is what this work aims to do.

During testing phase of program, some parts of the code may be rigorously tested but some parts might be neglected depending on the quality of test suite. The rigorously tested portions of the program would generally have fewer errors which are not fixed than the rest. Hence we can avoid analyzing the already tested portions of the program. This work aims to use the data from testing to direct a static analysis engine to the areas of unexplored code and the parts which have been tested with very less confidence.

Our approach takes the advantage of rigorous testing to perform better and efficient focused static analysis. The idea

Workshop on Dynamic Analysis (WODA 2005) 17 May 2005, St. Louis, MO, USA

[1]This work was supported in parts by a research grant from Microsoft.

is to use the coverage information to prune those paths of the program which have been sufficiently tested. By reducing the number of paths, we encourage the static analyzer to do more thorough analysis of the remaining code and explore more paths which it could not do earlier due to the large number of paths.

The proposed technique has useful application in the path insensitive static analysis. This includes abstract interpretation based checkers. Aggressive decisions can be made when the number of paths are reduced detecting additional errors which were not identified with all the paths criterion.

The nearest related work is Java Path Finder [6], a model checking tool for JAVA which uses runtime information to guide its model checking engine for detecting deadlocks. An attempt of visualization of failure points based on coverage was made in Tarantula [9]. In co-operative bug isolation project [10], the failure cases of program, given as feedback from user are used in efficient debugging. In contrast to these approaches, our work considers coverage information from testing to guide static analysis. This paper is organized as follows. In next section we describe our approach and algorithm, Section 3 contains some implementation details. Section 4 illustrates an example followed by some experiments and observations in Section 5. Section 6 contains conclusions and future work.

2. OUR APPROACH

There are two ways in which one can view the testing process. In one view, we can think that some paths are tested efficiently and bugs are revealed, leaving certain untested paths which have to be further analyzed for bugs. These paths are not tested because of improper test suite or rare conditions of execution for which test case(s) can't be easily generated. Hence we may put more effort on these paths rather than already tested paths. In the second view of testing, we may identify that certain control flow paths have always caused failure in the program. So the effort of static analysis can be levied on these paths. In this work, we focus on the first.

Our approach of using block coverage information from testing to drive static analysis consists of three phases as shown in Figure 1.

1. Mine the coverage information
2. Transform the control flow graph
3. Perform focused static analysis

During the testing process, we record the block coverage information which is then mined to identify the unexplored parts of the code in testing. The selection of blocks which have to be analyzed for possible bugs depends upon our block selection criterion, explained below. The program's control flow is reconstructed with pruning so as to include the blocks which are not properly tested. This step also identifies the blocks which might have been already explored but which are required in further analysis of unexplored blocks and the paths covering these blocks. In the third phase, we use this transformed program, and its control flow graph, for efficient and focused static analysis.

2.1 Block selection criterion

The efficiency of the approach lies in the selection of subset of blocks and paths using coverage data for a focused and

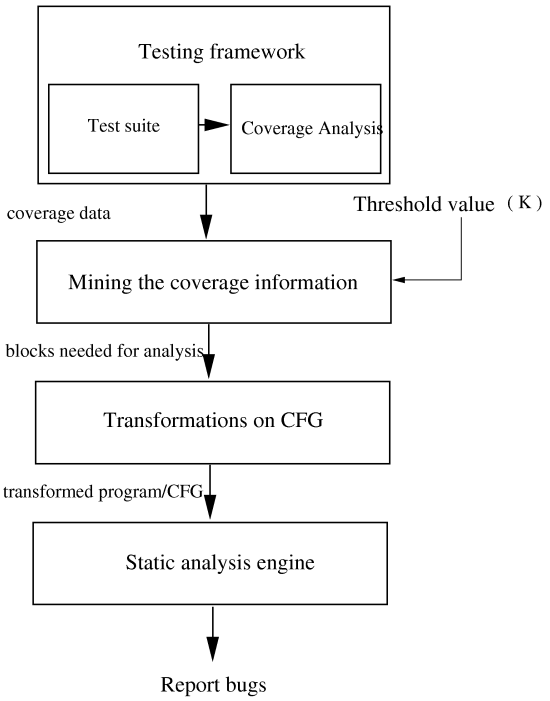

Figure 1: The three phases of our approach

thorough static analysis. We assume that the blocks which are substantially tested need not be analyzed further as the bugs might have been detected and fixed. A threshold value, 'K', is chosen for selection of the blocks to be analyzed. We can have two measures for 'K':

- If a block is covered in less than 'K' test cases ('K' is a threshold value, a parameter we choose), we would consider it for further static analysis. For the value of K=1, the set of blocks that are not executed in any of the test case are considered for static analysis.
- Alternatively, the value 'K' can be considered as the ratio of number of test cases executing the block to the maximum number times any block has been executed.

The result of path pruning is dependent on the value of 'K'. For small value of 'K', only uncovered paths are selected, while for large values of 'K', all the paths may be included for further analysis.

2.2 Transformation of control flow graph

The set of blocks tested with less confidence, B_a (subscript a indicates - to be *analyzed*), are identified based on the threshold value 'K'. These blocks are used for directing the static analysis engine. We need to get only those paths in the program which cover the set B_a. To do this, we reconstruct the program's control flow graph. According to the semantics of program, for each function there needs to be atleast one path from starting block to the returning/exit basic block. For each of the blocks in B_a, we construct the paths from the starting block to the terminating block, passing through any B_a and include each such block in the path for further analysis.

Let B_i (subscript i indicates - to be *included*) be the set of blocks to be included in analysis. The set of basic blocks B_p

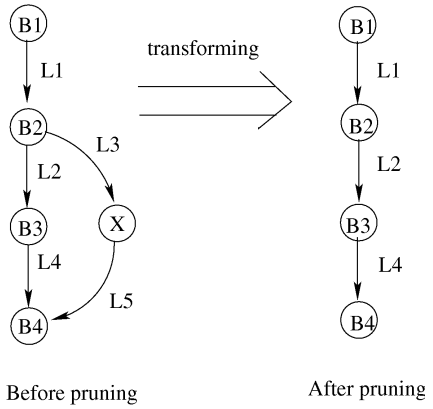

Figure 2: Transformation 1

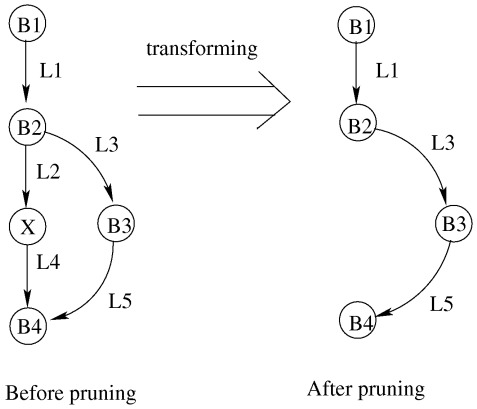

Figure 3: Transformation 2

(subscript p indicates - *pruned* blocks) which are not needed for the analysis of B_a are identified. Hence any control flow to one of these blocks, B_p, should be pruned. If B_n is the set of additional blocks to be included for analysis of B_a, B is set of all blocks in the program then the relation between these sets is as follows:

$$B_a \subset B_i$$
$$B_n = B - B_a - B_p$$
$$B_i = B_a \cup B_n$$

As we are reconstructing the paths for each block $bb \in B_a$, there would be no control flow resulting from bb to any of B_p and vice versa which needs to be analyzed. This is because, if there was to be a path from an element of B_p to bb, then it would have been included in B_i.

Let b be any block in the program's control flow graph. We have following three transformations for path pruning which are also illustrated in Figure 2 and Figure 3. These transformations are formulated with JAVA bytecode in view, which can be applied to other languages as well.

1. If b has any jump to an element from B_p then remove that jump i.e make it *null*.

2. If b has its immediate successor block in B_p, modify the conditional jump into unconditional jump to its right child.

3. Both left and right child of block b would not be in B_p, as we will be considering only those paths which would result in traversing uncovered paths. If this case was to arise then the block b itself would be in B_p.

In transformation 1, $B_a = \{B3\}$, $B_i = \{B1, B2, B3, B4\}$, $B_p = \{X\}$. The path $L1, L3, L5$ is substantially tested, leaving $L1, L2, L4$ for further analysis. Hence the jump instruction to X at $B2$ is replaced with *null* leaving the path $L1, L2, L4$.

In transformation 2, $B_a = \{B3\}$, $B_i = \{B1, B2, B3, B4\}$, $B_p = \{X\}$. The path $L1, L2, L4$ is substantially tested leaving $L1, L3, L5$ for further analysis. The conditional jump at $B2$ is replaced by unconditional goto to $B3$.

2.3 Correctness of transformations

The basic transformations on the control flow graph are additions (or modifications) and deletions of jump instructions so as to include only the unexplored blocks along with the paths resulting them. While doing this, the new program and hence its control flow graph should have a subset of paths from the original program without any extra paths. Also the paths should be complete i.e. there should be path from start to atleast one of the terminating basic blocks, and hence this path may be taken in one of the possible executions. To ensure these conditions, we generate complete control flow for each of the uncovered blocks and include all those blocks in the control flow path, which contain one of the uncovered blocks for focused static analysis. It is guaranteed that there is always a path to one of return/exit basic blocks. The three transformations will not cause undesired paths to be added. Moreover, there is no data flow information (incoming or outgoing) from the blocks included in our analysis to the pruned blocks. Hence the correctness of our approach is always preserved, while giving better results by aggressively analyzing fewer paths.

2.4 Advantages of the proposed approach

The advantages of the proposed testing driven, focused static analysis are:

- The paths which might not be analyzed during analysis of the entire code may be analyzed by our proposed approach which allows additional serious errors to be detected.
- More aggressive static analysis can be done with lesser number of paths, resulting in improved analysis of the code.
- The testing effort is leveraged for better static analysis.
- The runtime of static analysis can be reduced.
- The number of warnings issued as well as the noise should get reduced preventing serious errors from being missed by the programmer among false positives. This makes the framework more usable, practically.

The proposed method can be used in a comparative framework of static analysis before and after testing. Hence we

can focus on testing a few portions (which were reported from static analysis) and use testing results for making focused static analysis. So we will not miss any serious errors which can be reported by any of the frameworks.

The proposed approach has one potential limitation.

- We are relying on test suite for selection of paths for further analysis. We are emphasizing on specific unexplored paths which leads to incompleteness in other paths. This can prevent defects from getting caught in the paths that are pruned out. But this may not be serious concern, as those paths might have been already tested and hence the bugs in those paths would be revealed. However this shortcoming can be overcome either by analyzing the entire code before pruning or by analyzing the pruned portion separately.

This approach will not help increase the efficiency of the tools which does accurate and aggressive path sensitive analysis but, it would be helpful in the case of conservative static analysis. Even in the case of path sensitive analysis, where all paths are exhaustively analyzed, this methodology benefits in following ways:

- For the tools which consider only a subset of paths for analysis [2], our framework can act as a guide for selection of these paths.
- It gives the possible bugs in fewer, untested paths and hence we have lesser noise to deal with.
- The running time would be considerably reduced.

3. IMPLEMENTATION

Though our proposed approach is applicable for any language, tool and bug category, to verify the applicability and performance of our technique, we have implemented a prototype for JAVA. An ideal way to implement our technique is to use a coverage analyzer and a static analyzer which work on the same control flow graph representation. Hence we can force the static analyzer to look at specific paths deduced from the coverage analysis. As we were working with diverse tools, a uniform framework was not possible and hence in the current implementation, we have changed the JAVA class file so that it reflects the new program which has only those paths which result from our path pruning.

The basic algorithm for our proposed approach is shown in Figure 4. Let P be a program which needs to be verified. The program is instrumented (P') to record the block coverage. The test suite is prepared using which the program is tested. Since the test suites are not exhaustive, not all parts of the code are covered substantially. All the uncovered and less explored blocks B_a are identified from the block coverage information, which are used to build the program paths which need further analysis.

Control flow graph of the program P is generated. For each of the blocks in B_a, we construct the paths from starting block to the terminating block passing through one of the blocks in B_a. This is shown in step 4. Hence we retain the the paths in the program which are needed for the analysis of the unexplored basic blocks. Once this information is obtained, we will identify the basic blocks, B_p, which are not needed in further analysis of B_a. Hence the paths to B_p are pruned from future analysis. For this the class

```
1. Run the coverage analyzer.
2. Generate the CFG of the class.
3. For each function 'F'
   {
    For each block 'b' of 'F'
     If(cover[b] < K)
      list_uncovered[f].add(b)
   }
4. For each function 'F' such that
   list_uncovered is not empty
   {
    for each block b in list_uncovered[f]
    {
     Generate path(s) 'P1' from source block
           to the basic block 'b' in 'F'.
     Generate path(s) 'P2' from basic block
           'b' to the exit  block.
     Pf = P1 U P2
    }
    /* Pf is the list of blocks which
    need to be included in analysis */

    UnwatedBlocks[f] =  AllBlocks - Pf
   }
/* Now we have unnecessary blocks in the
   list UnwantedBlocks */
5. Scan the class file
   For each function 'F'
   {
    get UnwantedBlocks and remove the
    control paths containing these blocks.
   }
6. Do the static analysis of the
   transformed program using its CFG.
```

Figure 4: Algorithm for the process

file is modified by suitably replacing and/or adding jump instructions so that only necessary paths are present in the program. This is shown in step 5.

Finally, the modified classfile which reflects the program with pruned paths is given as input to static analyzer. In this prototype, we have used FindBugs as our static analysis engine. Entire process after testing is automated. The prototype is completely written in JAVA.

4. EXAMPLES

NULL dereference is one of the most frequent error committed by many programmers. Whenever an object is being dereferenced, it has to be initialized in all possible paths to the point of dereferencing. Missing the initialization in any of the path may cause this error on some input value which follows this control flow path.

There are many static analysis tools which detect possible NULL dereferences [7, 3, 4]. But the rate of false positives is generally high and hence a programmer may miss the occurrence of real bug amongst the noise. Hence we have considered this bug to prove the efficiency of our method. To reduce the overhead of analysis, FindBugs will leave the context information beyond a single branch. So the possible NULL dereference after the next branch can't be detected. The code fragment in Figure 4 contains a NULL dereference which is not reported by FindBugs.

It can be clearly seen that 's' is initialized to 'null' at

```
1     public void foo(int x, int y) {

2         NullDeref n = new NullDeref();
3         if (x<y)
4           s=null;
5         /*At this point we know s is null
           *in some path*/
6
7         if (x > y)
8           System.out.println("X > Y");
9
10        System.out.println("After if
          statement");
11
12        /* at this point we don't know
           * anything about the state of
           *'s'. So FindBugs won't generate
           * a warning here.*/

13        int mk = s.hashCode();
14        System.out.println("argv.length
                          2");
      }
```

Figure 5: Example of NULL dereference

statement 4. So if the path 2,3,4,7,10,13 is followed during some execution, then there is a potential NULL dereference at line number 13. As FindBugs does not carry the context beyond a single if statement, this bug is not detected by the tool. But if the code has been subjected to some test cases, then we have two scenarios.

1. x is greater than y
2. x is less than y

If the test suite was exhaustive then both the above cases would have been covered in testing and hence NULL dereference is identified. But if x < y is not tested, then this NULL dereference is not identified even in testing. We can use the partial test information to prune the tested path retaining unexplored paths. Doing this way, we would have neglected the intermediate branch i.e. the if statement along with its true branch is pruned. Hence for single path, we would test the program and detect the NULL dereference with FindBugs, which succeeds this time.

We believe that the applicability of the proposed approach can be easily extended for other bugs and tools whose analysis depends on number of paths. It is expected that our proposed approach would identify more defects with selective path pruning. Even when exhaustive, path sensitive analysis is employed, there is a great chance of missing the real bug amongst noise. If the bug in Figure 5 was present in a million lines of code, then among various other errors (along with noise), this bug could have been easily left unidentified. Reducing paths in a selective way, like this, may not miss the real bug.

5. EXPERIMENTS AND OBSERVATIONS

We have seen the applicability of our approach on a few programs of thousands of lines of code with FindBugs as static analysis tool. We have not yet tried the approach on other static analysis tools. The packages which we tested, pruned and subsequently analyzed with FindBugs are:

- *An instrumentation engine*: This program is a part of the coverage analyzer we have used to get statistical data.
- *Our prototype tool*: The entire implementation of our proposed approach
- *A few custom built programs*: These are smaller programs, containing known defects.

For each of the above, a test suite is built and some initial testing is done. From the results of testing, we have recorded coverage for each basic block and used that information to identify the unexplored blocks and the blocks having less confidence. The threshold value (K) for selection of the unexamined blocks is varied from 1 to 5. The analysis time for each of the program with pruning and without pruning is compared.

Our proposed approach reduces the number of paths to be analyzed and hence the time for analysis is observed to be lower. As we deal with fewer paths, noise is subsequently reduced and hence we have much lesser number of bugs to check for reality. Hence the burden on the programmer is reduced while probability of catching real bug is increased.

Let FP_1 be false positives without pruning and FP_2 be false positives with pruning. Let E be the total errors which are present in the program (real errors which may or may not be identified by any analysis tool), Er_1 be errors without pruning and Er_2 be the set of errors detected with our framework. Now for the subset of paths we are considering in our framework, let Er_s be the corresponding errors from Er_1. Let Er_n be the new errors detected from our framework. From our experiments we have observed that :

$$Er_s \subset Er_1$$
$$Er_2 \subset Er_1 \cup Er_n$$
$$Er_2 \subset E$$
$$FP_2 \subset FP_1 \cup FP_n$$

During the experiments, it came to us as a surprise that some noise FP_n, is added because of our framework. But this noise is comparatively negligible. The reason for the noise is the possibility of new paths being explored which was not done without our framework resulting in newer real bugs along with few false positives. As stated in Section 2.3, our framework does not add any new paths and programs behavior is not altered. The new paths explored which resulted in new real bugs, are part of all paths analysis which were ignored because of conservative static analysis, as is the case with FindBugs.

6. CONCLUSIONS AND FUTURE WORK

Coverage data from testing can be efficiently used to do a better and more efficient focused static analysis by pruning some unwanted paths for analysis. This approach reduces the time and space needed to analyze a program with improved error detection capability. Another notable advantage observed is that, the errors for subset of paths are less and hence these can checked without missing many real bugs amongst the noise, otherwise. Path insensitive analysis and the tools having conservative path assumptions will gain from the idea, and at the same time, additional uncovered errors may be explored. Of course, doing so means that the best we can do is increase the soundness. The issue

of completeness still remains but no program analysis technique strives to be complete because it is too hard. This approach looks to be viable for large softwares.

In this study, we applied the coverage analysis data from testing for efficient static analysis of the program. We are currently exploring more ways of using the results from testing for focused static analysis. One approach is to use failures to identify the buggy paths in the program. These paths can be subjected to exhaustive analysis.

It is also possible to have some runtime analysis along with coverage analysis to effectively pinpoint the suspicious points which would be later statically analyzed. We are also exploring how data from static analysis can be used to do more effective testing.

7. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[2] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, 2000.

[3] Cyrelli Artho. Jlint homepage: http://artho.com/jlint.

[4] D. Cok and J. Kiniry. Esc java homepage : http://www.cs.kun.nl/sos/research/escjava/index.html.

[5] D. Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, 1996.

[6] K. Havelund. Using runtime analysis to guide model checking of java programs. In *SPIN*, pages 245–264, 2000.

[7] D. Hovemeyer. Findbugs homepage: http://findbugs.sourceforge.net/.

[8] P. Jalote. *An integrated approach to software engineering*. Springer-Verlag New York, Inc., 1991.

[9] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 467–477. ACM Press, 2002.

[10] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Dec. 2004.