

A Machine Learning Approach to Finding Bugs

Fancong Zeng
Department of Computer Science
Rutgers University
Piscataway, NJ, 08854
fzeng@cs.rutgers.edu

ABSTRACT

Recently a number of tools have been developed for finding bugs in programs without executing the programs. While these tools are helpful, they often report too many false alarms, false negatives or both. This paper does not present any new such tools. Rather, it is focused on providing a machine learning approach for integrating existing bug finding tools into a practical and adaptive toolset.

1. INTRODUCTION

It is desirable to find bugs in large programs without actually executing these programs. Recently different research groups have developed a number of bug finding tools using various static analysis techniques such as model checking, theorem proving and dataflow analyses.

[8] compares some of these tools and reports some initial yet interesting findings. Unlike the methodology used in [8], I examine some bug finding tools in the context of testing and debugging processes. These processes have several important properties.

One of these properties is that static bug-finding tools are expected to find bugs which are difficult to be caught by traditional test cases, e.g., security vulnerability and timing-related bugs like deadlocks. Another is that software testers find and report bugs using these bug finding tools among others and software developers introduce and fix bugs. Yet another is that there are (human resource) costs associated to finding and fixing bugs.

Thus it is important for testers not to report to developers too many false alarms regarding hard-to-catch bugs. It is also important that the bug finding tools do help shake out some hard-to-catch bugs.

To achieve the above two goals, there should be a good way to use the static bug-finding tools. Suppose each bug finding tool is considered as an expert, the problem of utilizing mul-

iple tools can be reduced to the problem of “predicting with expert advice”, a well-known machine learning problem [3].

In this paper, I formulate a variant of the “predicting with expert advice” problem and devise a practical solution to it, so that software engineers can follow this solution in their everyday bug finding and fix activities.

Outline The rest of the paper is organized as follows. Section 2 introduces some popular bug finding tools via static checking and uses a deadlocked Java program to show the imperfection of these tools. Section 3 defines a variant of the “predicting with expert advice” problem with each bug finding tool as an expert. Section 4 presents a practical solution to the problem. Section 5 discusses the solution. Section 6 concludes the paper. Section 7 lists the references.

2. BACKGROUND

In this section, I briefly describe three well-known static tools and then use a simple deadlocked Java program to demonstrate the usage and limits of these tools. Deadlocks are a timing-related bug which is difficult to catch and repeat.

2.1 Three Tools

ESC/Java This tool uses a theorem prover to verify that code matches specifications. Generally programmers supply specifications in terms of annotations to the source code. In some cases, programmers do not need to specify annotations and ESC/Java checks some default properties like deadlocks [5].

JLint This tool operates on bytecode and exploits inter-procedural dataflow analysis and some syntactical checks to find bugs and coding pitfalls. In particular, JLint builds a lock graph and signals deadlock warnings if there is a cycle in the graph [1].

FindBugs This tool works at the bytecode level and relies on bug patterns to find bugs. It favours efficient analyses, so it does not use expensive inter-procedural dataflow analyses. Consequently, FindBugs does not report deadlocks effectively [7].

2.2 Deadlock Detection

Figure 1 shows a simple deadlocked Java program. This program simulates money transfer between accounts. There

```

import java.util.Random;
public class Transfer {
    public static final int NumberOfAccounts = 20;
    public static final int InitialFund = 1000;
    public static final int MaxFund = 1000000;
    public static final int NumberOfThreads = 10;
    private static Object[] Accounts= new Object[NumberOfAccounts];
    private static long[] balance = new long[NumberOfAccounts];
    public static void main(String[] a) {
        for (int i=0; i<NumberOfAccounts; i++) {
            balance[i] = InitialFund; Accounts[i] = new Object(); }
        for (int i=0; i<NumberOfThreads; i++){
            TransferThread trans = new TransferThread();
            new Thread(trans).start(); }
    public static boolean doTransfer(int f, int t, int a) {
        synchronized (Accounts[f]) {
            synchronized (Accounts[t]) {
                if (balance[f] < a){
                    System.out.println("Transcation Aborted: insufficient fund.");
                    return false;}
                if (balance[t] + a > MaxFund){
                    System.out.println("Transcation Aborted: too much fund.");
                    return false; }
                balance[f] -= a;
                balance[t] += a;
                System.out.println("Transcation Completed: transferred "
                    +a+" dollars from "+f+" to "+t);
            }
        }
        return true;}
    public static class TransferThread implements Runnable{
    public synchronized void run() {
        while (true) {
            int fund = Math.abs(new Random().nextInt()) % InitialFund;
            int source = Math.abs(new Random().nextInt())
                % NumberOfAccounts;
            int dest = 0;
            do { dest = Math.abs(new Random().nextInt())
                % NumberOfAccounts; }
            while (dest == source);
            doTransfer(source, dest, fund);
        }
    }
}
}
}
}

```

Figure 1: A simple deadlocked Java program

are twenty accounts and ten threads. Given any two different accounts, a thread is used to transfer money from one account to the other. For the transfer to begin, the thread has to acquire the locks for both accounts. In the program showed in Figure 1, deadlock occurs when two or more threads are involved in a cyclic wait for locks.

FindBugs does not have expensive analyses to support static deadlock detection, so it cannot detect the deadlock in the program. On the other hand, JLint and ESC/Java report some warnings for this deadlock problem.

If the “ $if(balance[f] < a)$ ” block were moved to the place between the two synchronized statements, there would still be a deadlock problem in the code, but in this case JLint would not report a warning. This is a false negative example.

If the doTransfer method were synchronized, then there would not be any deadlock problem. But in this case, both JLint and ESC/Java would still report a deadlock warning. This is a false positive example.

Actually, without the aid of annotations, ESC/Java often

produces too many false positives of deadlocks so that by default it does not report deadlock warnings. For the study, I turned on the flag to have ESC/Java report deadlock warnings.

The above example illustrates that static bug-finding tools are helpful but they suffer from false alarms, false negatives, or both.

3. PROBLEM

Consequently, there are a lot of question-marks regarding using these tools to help find and remove hard-to-catch bugs in practice. Example questions are: suppose a tool reports a warning about a deadlock in a large program, should the tester report the problem to the developer or not? How should the developer deal with the report? Should the developer spend time finding the root cause of the reported deadlock which may be just a false positive? Should the tester push the developer to remove the potential deadlock?

To address the above questions, I first formulate a problem for using these tools.

Suppose there are N tools, each of the N tools is considered as an expert. The tester can also be considered as an expert. The problem of using these tools can be formulated as an variant of the “Predicting with Expert Advice” problem.

Specifically, whenever the tester is given a bug type and a code region, he gets each expert to report “yes” or “no” to the question whether the code region contains an instance of this bug type. Suppose at least one expert says “yes”, a predictor uses the N (or $N+1$ if the tester also considers himself an expert) expert outputs to make its own prediction of “yes” or “no.” Then, under some conditions, the tester pushes the developer to debug and to get back to him with a true answer w.r.t. whether the code region contains the bug.

To be general, I make no assumption of the independence or the quality of the tools. Thus, it is infeasible to achieve absolute quality in the prediction. Instead, a practical goal that the predictor performs nearly as well as the best tool so far. I use the number of mistakes made so far to compare the performance, and I denote this goal as the Testing Goal. This goal is reasonable because in practice given a collection of tools we generally do not know which tool is the best. Furthermore, it is often the case that different tools perform best for different bug types.

Another practical goal is that the tester pushes the developer to do the debugging work at a reasonable rate, which is called *debug rate* in the rest of the paper. I denote this goal as the Debugging Goal, and this goal is to strike a good balance between real bug fixing and wasted efforts. Note that the Testing Goal and the Debugging Goal are related. In addition to solving the “Predicting with Expert Advice” problem, the next section will examine the relation between the two goals.

4. SOLUTION

There are many algorithms to solve a “Predicting with Expert Advice” problem. For the problem I formulated in the

previous section, I devise a solution inspired by and based on an algorithm described in [4] for the problem.

For every bug type, the tester uses an implementation of the following procedure to report warning for bugs of that type and to push the developers to debug. The procedure exploits N tools, uses two parameters: $\eta > 0$ and $0 \leq \epsilon \leq 1$, and initialize the weights of all experts at round 1, denoted as $W_1 = (W_{1,1}, \dots, W_{N,1})$, to vector 1.

For each round $t = 1, 2, \dots$

1. Apply the set of N tools to code region C_t , and get an output $I_t = (I_{1,t}, I_{2,t}, \dots, I_{N,t})$, where $I_{i,t} = \text{“Yes”}$ or “No” . “Yes” means that there is a bug warning for C_t .
2. if I_t does not contain a “yes” , then let $W_{t+1} = W_t$ and go to the next round. Otherwise the predictor outputs $I_{i,t}$ with probability $W_{i,t} / \sum_{j=1}^N W_{j,t}$, $i = 1, \dots, N$.
3. Draw a Bernoulli random variable Γ_t s.t. $P[\Gamma_t = 1] = \epsilon$.
4. If $\Gamma_t = 0$, then let $W_{t+1} = W_t$, and go to the next round. Otherwise push the developer to figure out whether there is a bug in C_t and to remove the bug if it does exist.
5. Use R_t to denote the answer given back by the developer, and assume R_t is correct. If $R_t = I_{i,t}$, then set $W_{i,t+1}$ to $W_{i,t}$, else set $W_{i,t+1}$ to $W_{i,t}e^{-\eta/\epsilon}$. Go to the next round.

One nice property of the above procedure is:

Fix $n > 0$, the number of times that not all tools say “no” . Let $\epsilon = m/n$ and $\eta = (\sqrt{2m \ln N})/n$, and let $M_{P,n}$ denote the number of *mistakes* — either false positives or false negatives — made by the predictor, and $M_{B,n}$ denote the number of *mistakes* made by the best tool, then

1. The expected number of times that the tester pushes the developer to debug is m .
2. $E[M_{P,n} - M_{B,n}] \leq n\sqrt{2 \ln N / m}$.

The above 2 claims can be proved by adapting the proof of Theorem 1 in [4].

Furthermore, [6] shows that the optimal result of $E[M_{P,n} - M_{B,n}]$ is $\Theta(\sqrt{n})$ for a $\text{“Predicting with Expert Advice”}$ problem, and [4] proves that a *debug rate* of order $(\ln n)(\ln \ln n)^2/n$ is sufficient for $E[M_{P,n} - M_{B,n}]$ to grow sub-linearly with probability 1.

5. DISCUSSION

The main difference between my work and other work using machine learning in finding bugs is that my work is to use existing bug-finding tools in a long-term software process, while others like [2] are to use machine learning to design new bug finding tools. To some extent, those new bug finding tools, when they become practical, can be incorporated to my proposed solution.

Different software teams may customize this solution to fit into their needs. For example, for deadlock problems, I consider myself an expert, and using the above process I mainly rely on my expertise to report deadlock bugs and adjust my behavior based on the feedback.

One customization is that testers may treat false positives and false negatives differently. To implement this, testers can use two ϵ 's: $0 \leq \epsilon_1 < \epsilon_2 \leq 1$. Specifically, ϵ_1 (resp. ϵ_2) is applied when the prediction is “No” (resp. “yes”) in step 3. Compared to using ϵ_2 in both cases, the debug rate would be less but there would be more prediction mistakes.

Another customization is to drop tools of bad performance so far and/or to add new tools at some point and then to restart the procedure with a normalized weight initialization. The new tools, if any, are weighted 1 too. A bound for regret in this case is under investigation.

This solution is easy to implement, and I am in the process of implementing it within Eclipse. It is also in progress to collect data to support and to polish the approach for software engineers' everyday use.

6. CONCLUSION

This paper provides a novel and promising machine learning based approach to finding bugs. Extensive experiments to evaluate the approach are in progress.

7. REFERENCES

- [1] C. Artho. *Finding faults in multi-threaded programs*. Master's thesis, Institute of Computer Systems, Federal Institute of Technology, Zurich/Austin, 2001.
- [2] Y. Brun and M. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering.*, 2004.
- [3] N. Cesa-Bianchi, Y. Freund, D. P. Helmbold, D. Haussler, R. E. Schapire, and M. K. Warmuth. How to use expert advice. *Journal of the ACM*, 44(3):427–485, 1997.
- [4] N. Cesa-Bianchi, G. Lugosi, and G. Stoltz. Minimizing regret with label efficient prediction. *IEEE Transactions on Information Theory*, TO APPEAR.
- [5] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for java. In *Proc. PLDI*, 2002.
- [6] J. Hannan. Approximation to bayes risk in repeated plays. *Contributions to the Theory of Games*, 3:97–139, 1957.
- [7] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.
- [8] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *The 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04)*, November 2004.