On the Localization of Branch Faults in Object-Oriented Program*

Xinming Wang
Department of Computer Science & Engineering
Hong Kong University of Science & Technology
Clear Water Bay, Kowloon, Hong Kong, China
rubin@cse.ust.hk

S.C. Cheung**
Department of Computer Science & Engineering
Hong Kong University of Science & Technology
Clear Water Bay, Kowloon, Hong Kong, China
scc@cse.ust.hk

BRIEF DESCRIPTION

Problem: To locate faults that reside in branch statements. Approach: To perform fault localization using program state history.

ABSTRACT

A common type of program fault is related to branch statements, which is referred to as *branch fault*. In an object-oriented (OO) programming paradigm, branch statements are typically used to implement state-dependent behavior of classes or clusters of classes. Thus, one effective way to locate the branch fault in an OO program is to identify a condition on the object or object cluster state that activates the fault. In this paper, we briefly present our recent work regarding this basic idea. The presented approach combines static and dynamic analysis. It first generates a set of suspicious candidates by extracting relevant predicates from the source code, and then filters out irrelevant candidates by analyzing program state history which is recorded *incrementally* in the executions of successl and failure-inducing test cases.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Testing and Debugging – *debugging aids, diagnostics*

General Terms

Verification, Measurement

Keywords

branch fault, fault-enabling condition, dynamic dominance tree

1. INTRODUCTION

Software debugging is a process that locates and removes faults in a program. It involves at least two core activities: finding the cause of a failure (known as a fault) and modifying the related code. Previous empirical studies [7] report that fault localization if performed manually often contributes to major software development cost. Automating the process of fault localization is therefore desirable.

Due to the prevalent use of branch statements in programming, faults related to them have received extensive attention [5] [8] [9] [15]. We refer to this kind of fault as a *branch fault*. Branch faults are especially common in object-oriented (OO) program. Many OO design methodologies encourage the use of state models to analyze state-dependent behavior of classes or clusters of classes [1].A vastly popular way to manually implement them is to use branch statements. Branch faults are thus introduced into the system by mistake or misunderstanding of these behaviors. For example, in the entire bug history of a medium-sized java program *nanoxml*[1], 75% of bugs are branch faults.

For brevity, we classify the branch faults into three main categories. Their distribution in *nanoxml* is shown in Table 1. Other branch faults exist but are not as common as these three. Thus, in this work we focus on these three types only. Some examples of these three categories in OO program are illustrated in Table 2.

1. **Erroneous expression**: The conditional expression of a branch statement is incorrect. Thus, in some case the program execution will follow a wrong branch.
2. **Erroneous branch**: The computation in a branch statement is faulty. Whenever this branch is executed, the error may thus be triggered.
3. **Missing branch**: An entire branch is missing. This branch can be the "else" part of the "if" statement, the whole "if" statement, or a case in "switch" statement, and so on.

Table 1: Distribution of branch faults in *nanoxml*

Erroneous expression	Erroneous branch	Missing branch	Others (Not branch fault)
15%	42%	18%	24%

The branch fault is activated if the program state satisfies some particular condition prior to the execution of the branch statement. We refer to this condition as the *fault-enabling condition*, which intuitively is a predicate over the program state. For example, for an "Erroneous condition expression" fault, the fault-enabling condition is the exclusive-or of correct expression and erroneous expression. Some concrete examples of fault-enabling conditions are shown in Table 3.

We conjecture that locating a branch fault should involve finding the erroneous or missing branch statement in the program as well

FSE'06, Nov 5–11, 2006, Portland, Oregon, USA.

* This research is supported by a grant from the Research Grant Council of Hong Kong (Project No: HKUST6187/02E).

** Dr. S.C. Cheung is the academic advisor of Xinming Wang.

[1] Including 5 program revisions with totally 33 recorded bugs. It is retrieved from SIR [3] repository.

Table 2: Examples of branch faults in AspectJ compiler (all from class org.aspectj.weaver.bcel.BcelWeaver)

Bug ID	Category	Bug fix
125699	Erroneous conditional expression	Change "if(advice.getConcreteAspect().isAnnotationStyleAspect()){…}" at line 502 to "if(advice.getConcreteAspect().isAnnotationStyleAspect()|| advice.getDeclaringAspect()!=null&& advice.getDeclaringAspect().resolve(world).isAnnotationStyleAspect()) {…}"
77166	Erroneous branch	Modify the branch body of "if (!ba.hasMatchedSomething()) {…}" at line 1004
119882	Missing branch	Adding a branch statement at line 1092: "if(theDelegate.getClass().getName().endsWith("EclipseSourceType")){...}"

as the fault-enabling condition. This information facilitates fault repair. When the complete specification of the program is not available, localization of branch faults by static analysis would be difficult. Thus, some authors propose dynamic analysis approaches, such as DIDUCE [6] and Carrot [12]. The basic intuition is to first hypothesize a list of suspicious program points. They are typically located right before every branch statement. Next, the history of the program state is recorded in both the successful runs and failure-inducing runs. Then an analysis is performed to find a list of predicates that hold and only hold in the execution of failure-inducing test cases. These predicates form a candidate set of suspicious fault-enabling conditions. Developers can thus select a condition from the candidate lists. If the condition is a fault-enabling condition, the branch fault is located.

Table 3: Examples of fault-enabling conditions

Faults	Fault-enabling Condition
Wrongly type "if(A||B)" as "if(A&&B)"	(¬A&&B)||(A&&¬B)
Processing in "if(A&&B){…}" is erroneous	(A&&B)
Missing the else part of "if(A&&B){…}"	(¬A||¬B)

Although the basic idea is straightforward, in practice there are at least three important issues. First, pre-determining a set of relevant predicates is required. Existing approaches use a fixed scheme set to generate such a set. For example, for any two integer variables a and b, $a>b$, $a<b$, $a=b$, $a\leq b$, $a\geq b$ are added into the initial set. However, it is heuristic and thus may not cover the fault-enabling condition, which seriously degrades the fault-locating accuracy [12]. On the other hand, it is practically impossible to include all predicates on the program state. Hence, how to find a suitable initial set becomes an important issue. Second, recording program state history is a very expensive operation in terms of space and time. For example, running the Daikon [4] tool (which is used by Carrot to record program state history) on one single run of a utility java program will result in 11.5 gigabytes of program states history [11]. Scalability thus is a prime concern. Finally, a lot of predicates in the candidate set are redundant, which are undesirable since it increases the analysis cost. This problem is also discussed in [8].

In this paper, we focus on how to alleviate the first two problems of localizing faults in OO programs. In an OO program, branch statements are often used to implement state-dependent behavior of classes or class clusters. Thus, the conditional expression and its corresponding fault-enabling condition would involve querying the state of an object or object cluster. Based on this observation, we propose a static analysis step to find relevant predicates from a source code instead of using a fixed set of a predicate scheme. We also investigate how to exploit the relationships between program points and how the program's state history can be recorded incrementally in order to improve the time and space efficiency.

The remainder of this paper is organized as follows: Section 2 discusses the related work for locating branch faults. Section 3 analyzes the problem in more detail while Section 4 presents our approach. Section 5 discusses how this work will be evaluated, and portrays future work.

2. RELATED WORK

Various studies have been conducted on automated fault localization. Static analysis [10] checks a source code against several predefined properties. Violation of these properties indicates potential faults. Dynamic analysis, instead, relies on the information collected from program execution, which is more suitable for detecting branch faults that are related to program-specific semantics. Besides DIDUCE [6] and Carrot [12], there are at least two other categories of dynamic analysis techniques that can be used for locating a branch fault:

Statistical debugging: statistical debugging [5] [8] [9] focuses on the comparison of statistics that characterize the runtime behavior between failure-causing executions and successful executions of a program. The runtime behavior is recorded in the evaluation history of conditional expression as a predicate. Statistics on this history are collected over multiple executions and analyzed afterward. For example, Chao et al. [9] build two statistical models for each predicate on evaluation history of failure-causing executions and successful executions, respectively. If the two modes of a predicate differ significantly, then this predicate is suspicious. A common requirement for statistic debugging is that a large amount of test cases, both successful and failure-inducing, are needed to gain statistical significance. In practice it may not be satisfied, especially for failure-inducing test cases. Another shortcoming is that it only gives the location of suspicious conditional statement without the information on the fault-enabling condition.

Predicate switching: predicate switching [15] follows a totally different idea from statistical debugging. Instead of just observing and comparing the dynamic behavior, it tries to modify the runtime behavior of a program by changing the evaluation results of a predicate's outcome and alters the control flow. Given a failure-inducing test case, this approach repeatedly executes the program and tries switching the results of each predicate evaluation instance in a separated run until a run of the program gives the correct result. Then, by examining the last switched predicate, the faulty conditional statement can be identified. This idea is similar to Delta Debugging [14], which tries to change the value of variables instead of the predicate evaluation result. The advantage of predicate switching is that it only requires one failure-inducing test case to locate the fault. However, predicate switching can only handle an erroneous conditional expression. In case of erroneous processing, switching the corresponding predicate evaluation result generally does not produce the correct output. Besides, a fault related to a missing branch can not be located by this technique.

3. ANALYSIS OF THE PROBLEM

As mentioned in Section 1, one effective way to locate a branch fault is to find a list of candidate fault-enabling conditions that hold and only hold in failing runs of programs. To find this candidate set, we need to first hypothesize a list of relevant predicates and then use dynamic analysis to filter out the irrelevant. Two basic observations in an OO program are helpful for finding such a relevant predicates set:

First, in an OO program, conditional expression of branch statements are typically predicates on the *observable state* of objects instead of an *internal state*. This observation is not surprising since OO methodologies encourage information hiding and discourages direct examination of the value of object attributes to determine the program state. For example, in Table 2, all conditional expressions consist of predicates on the return value of query-only methods [13] instead of object attributes. A more complicated case involves the observable state of a cluster of objects, such as whether two list objects contain the same number of items.

Second, the conditional expression often consists of several atomic predicates over the observable state of an object. These atomic predicates are often meant to be the *predicate abstraction* of observable states and we observe that they often appear more than once in the source code. Let us clarify the meaning of atomic predicates using an account example. Suppose that the observable state of a class account contains an integer attribute representing its current balance. Without other information, guessing the relevant predicates on it would be difficult. However, if the balance is compared with some constant values such as "100" and "500" throughout the source code (e.g., balance > 100, balance < 500, we call them atomic predicates), then the relevant predicates of the fault-enabling condition can be heuristically constructed as a logic expression of these atomic predicates (e.g. balance > 100 and balance < 500). Our observation is that in an OO program, this is often the case. Although predicate abstraction of an observable state may not be explicitly available in the class definition, they can be mined from other parts of source codes containing reference to the observable state.

These two observations give hints on how to determine the set of hypothesized fault-enabling conditions: for each class, we first identify its observable state from its definition, then we mine the atomic predicates to generate the initial relevant predicates set. However, to what extent these observations hold for typical OO programs is unclear. We are implementing a static analysis tool prototype to examine several large open-source programs to verify the above conjecture.

Now assume that given a program point, we can construct a relevant predicate sets as candidates of fault-enabling condition. The next research question is where a fault-enabling condition may locate in the source code. For an erroneous conditional expression fault and an erroneous branch fault, the position is at the conditional expression of a branch statement. However, for a missing branch fault, the position is unclear. For simplicity, we choose the entry and exit of block statements, where the missing branch may most likely be located.

4. APPROACH

Based on the analysis of the problem, we propose our approach for locating a branch fault in an OO program. In this section, we first formulate the problem, then introduce the basic work flow of our approach and discuss our techniques.

4.1 Formulating the problem

Based on the analysis in Section 3, we define relevant concepts as follows:

Definition 1 (Observable state): Given an object of class C, its observable state with respect to a concrete state S is defined as $O = <O_1, O_2, O_3 \ldots O_n>$. Each O_i is called an observable state component which is determined by a query-only method M of C that does not receive object reference as argument. If M returns a primitive type value, then this return value directly forms a component. If it returns an object reference of Class C', then the observable state components of C' are added as observable state components of C.

Given m objects of classes $C_1, C_2, C_3 \ldots C_m$, the observable state of this m-objects cluster is the union of all these objects' observable states in addition to some joint state components. These joint state components are determined by query-only methods of $C_1, C_2, C_3 \ldots C_m$ that receive object references of $C_1, C_2, C_3 \ldots C_m$ as arguments.

Note that an observable state may contain a recursion structure. For example, suppose there an item object has a query-only method returning reference to its container, while its container in turn has a pure method returning reference to this object.

Definition 2 (Predicate abstraction): A predicate abstraction p is defined as an atomic predicate on an observable state O of an object or an object cluster. This can be a comparison of state components with some constant value, a comparison of two state components, or the state component itself if it is a Boolean value.

Given a program point P, there are a set of objects visible at the program point. We model the *program state* at P as the observable state of these visible objects. The relevant predicates at this point are thus a logic composition of predicate abstractions on this observable state.

4.2 Work flow of our approach

The basic work flow of our approach is: For each program point, we generate a set of relevant predicates. Then we run the failure-inducing and passed test cases separately and record an observable state history for all program points. Next, in the relevant predicates set of each program point, we identify predicates that hold and only hold in the execution of failure-inducing test cases. Finally, developers review these predicates to find the real fault-enabling condition.

In the first step, we use a static analysis technique to find predicate abstractions in the program and use them as atomic predicates of candidate relevant predicates. If the observations in Section 3 hold, then intuitively the resultant predicate set would have a higher chance to cover the fault-enabling condition than the aforementioned heuristic approach.

The second step is to dynamically collect program state history. While it can be done automatically, the time and space complexities become the main concerns. Specifically, a straightforward implementation that retrieves and records whole program states at every execution instance of a program point is impractical for real-life programs like AspectJ compiler, which contains thou-

sands of methods with tens of thousands of program points. Each program point may be executed many times in a run. Recording a full program state history is neither time nor space-efficient.

To tackle these problems, we observe that there are a lot of redundancies in the straightforward implementation. Some objects are shared among different program points either intra-procedurally or inter-procedurally. And not all components of the observable state change across two consecutive program points sharing this object. A more appealing approach is to record a full observable state only at some program points. Then at other program points we only record state delta, that is, components that have changed afterwards. Intuitively, two object references at two program points may refer to the same object. We say that object reference R_A at program point A covers object reference R_B at program point B iff: at an execution instance of B, R_B refers to object o, and then there must exist previously an execution instance of A in which R_A also refers to o. Once we determine the covering relation of all object references at different program points, the recording process could be simplified. For those objects not covered by other object references, we record all their object states; otherwise we only record the state delta. However, to determine the exact covering relation would be difficult. To this end, we introduce the notion of a *dynamic dominance graph* to determine a safe approximation of the covering relation. The basic idea is to determine the dynamic dominance relation between program points, and then use static analysis to determine a subset of covering relations.

A remaining issue is how to find the fault-enabling condition. This is a standard AI problem setting which is similar to those focused on inductive logic programming (ILP) [2]. We are searching for an effective algorithm for this particular problem.

5. EVALUATION

Our experimental plan aims to investigate whether our approach can efficiently locate a branch fault in real-life programs. The evaluation is three-fold. First, whether or not the fault-enabling condition can be covered by the relevant predicate set generated in the static analysis step. Second, whether or not the incremental recording approach can improve the time and space efficiency comparing to the straightforward approach. Third, how the quantity and quality of passed and failure-inducing test cases affect the accuracy of the results. Currently, we are conducting an experiment on several open-source projects including AspectJ to gather raw data for analysis.

There is still a lot of work to be done. In particular, it is unclear whether or not using an abstract program states to hypothesize fault-enabling conditions is effective for locating branch faults. More careful investigation on bug history of real-life projects is desirable. Besides, effective algorithms for finding fault-enabling conditions from recorded program state history is still to be devised. We are also interested in investigating the possibility of running this algorithm online.

6. ACKNOWLEDGMENTS

The authors wish to thank Dr. W.K. Chan for his insightful comments.

7. REFERENCES

[1] Briand, L. C., Penta, M. D., and Labiche, Y. Assessing and Improving State-Based Class Testing: A Series of Experiments. *IEEE Transactions on Software Engineering 30(11),* pp. 770-793, Nov 2004.

[2] Cohen, W. W. Grammatically Biased Learning: Learning Logic Programs Using an Explicit Antecedent Description Language, *Artificial Intelligence 68*, pp. 303-366, Aug 1994.

[3] Do, H., Elbaum, S., and Rothermel, G. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal 10(4),* pp. 405-435, Oct 2005.

[4] Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering 27(2),* pp. 99-123, Feb 2001.

[5] Fei, L., Lee, K., Li, F., and Midkiff, S. P. Argus: Online Statistical Bug Detection. *Proceedings of Fundamental Approaches to Software Engineering,* Vienna, Austria, Mar 2006.

[6] Hangal, S. and Lam, M. S. Tracking down software bugs using automatic anomaly detection. *Proceedings of the 24th international Conference on Software Engineering*. Orlando, USA, May, 2002.

[7] Ko, A. J., Aung, H. H., Myers, B.A. Eliciting Design Requirement for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks. *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, USA, May 2005.

[8] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. Scalable Statistical Bug Isolation. *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*. Chicago, IL, USA, Jun 2005.

[9] Liu, C., Yan, X., Fei, L., Han, J., and Midkiff, S. P. SOBER: Statistical Model-based Fault Localization. *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th International Symposium on Foundations of Software Engineering,* Lisbon, Portugal, Sep 2005.

[10] Musuvathi, M., Park, D. Y., Chou, A., Engler, D. R., and Dill, D. L. Cmc: A pragmatic approach to model checking real code. *Proceedings of the 5th Symposium on Operating System Design and Implementation,* Dec 2002.

[11] Perkins, J. H. and Ernst, M. D. Efficient Incremental Algorithm for Dynamic Detection of Likely Invariants. *Proceedings of the 12th International Symposium on Foundations of Software Engineering*, Newport Beach, USA, 2004.

[12] Pytlik, B., Renieris, M., Krishnamurthi, S., and Reiss, S. P. Automated fault localization using potential invariants. *AADEBUG,* Sep 2003.

[13] Salcianu, A. and Rinard, M. Purity and Side Effect Analysis for Java Programs. *Proceedings of 6th International Conference on the Verification, Model Checking, and Abstract Interpretation,* Paris, France, Jan 2005.

[14] Zeller, A. "Isolating cause-effect Chains from Computer Programs", *Proceedings of the 10th International Symposium on Foundations of Software Engineering*, Charleston, US, Nov 2002.

[15] Zhang, X., Gupta, N., and Gupta, R. Locating Faults Through Automated Predicate Switching. *Proceedings of the 28th International Conference on Software Engineering,* Shanghai, China, May 2006.