

Fault Insertion in Concurrent Object-Oriented Programs for Mutation Analysis and Testability Measurement

Sudipto Ghosh

Computer Science Department, Colorado State University
ghosh@cs.colostate.edu

1. Introduction

Testability measurement and mutation analysis approaches require programs to be seeded with faults that represent plausible programmer errors. Software testability [5] is defined as the likelihood of a program failing on the next test input from a predefined input distribution, given that there is a fault in the program. Mutation analysis [2] requires the insertion of faults into a program with the goal of creating mutation-adequate test sets that distinguish the modified program (*mutant*) from the original program.

There is a lack of methods and techniques for measuring testability and performing mutation analysis of *concurrent* object-oriented programs. An approach towards fault insertion into concurrent Java programs is proposed in this paper. The approach is based on two methods: (1) mutation of the program based on keywords, and (2) creation of mutants based on conflict graphs created by performing static analysis of the code. The conflict graphs are similar to those described in DU-path based approaches [6] (e.g., Parallel Program Flow Graphs and Trace Flow Graphs) that use a graph representation of the concurrent processes (or threads).

2. Keyword-based approach

Java provides two syntactic forms for synchronization. These forms are based on the `synchronized` keyword and can be used for blocks or methods. Block synchronization takes as an argument the object to lock and allows any method to lock any object. Method locking serializes the execution of the method. There are several rules for synchronization (e.g. in an inner class, inheritance, synchronization within constructors, locking static fields; see [4] for details). We focus on the cases where access to shared data needs to be protected, but is not, i.e., programmers may not use the `synchronized` construct where they should. These errors may arise because of poor design, carelessness, or lack of knowledge about synchronization rules in Java.

We define a pair of operators `RSYNCM` and `RSYNCB` related to the removal of the keyword `synchronized` from

methods and blocks respectively. To increase the likelihood of killing mutants it may be necessary to apply different sequences of methods in addition to enhancements in the test cases themselves. Equivalence of mutants is impacted by the interleavings. Timing considerations are important. The time slice given to each thread is not under the control of the tester and can lead to different interleavings with different results. The number of interacting threads needs to be controlled by the tester. Since each program has at least two threads, the tester needs to run the original program and the mutant with at least two threads each.

3. Program conflict graph-based approach

Given the source code of the program, we can construct a conflict-graph that depicts how different threads may interact. The conflict graph is actually a multi-graph built from individual control-flow graphs for each thread executing an interacting method. There are cross edges between the graphs to indicate where there may be a conflict between the nodes. Based on the two kinds of synchronizations possible in Java, there may be three basic kinds of graph combinations: (1) method and method, (2) block and block, (3) block and method.

Consider the following method `withdraw` that has an attribute called `balance` which is a shared variable for several methods.

```
1... public synchronized void withdraw(int amt) {
2...     if (balance > (amt+TRANSACTION_FEE)) {
3...         balance = balance - amt - TRANSACTION_FEE;
4...     } else {
5...         balance = balance - TRANSACTION_FEE;
6...     }
7...     return;
8... }
```

Since the `synchronized` keyword is used, the code is safe. However, if it was not used, there would be potential problems depending on the interleaving of the operations related to reading and modification of the value of `balance`.

Assuming that there are only two threads and both execute the `withdraw` method, we draw a graph that shows the potential conflicts between the definition and use pairs of

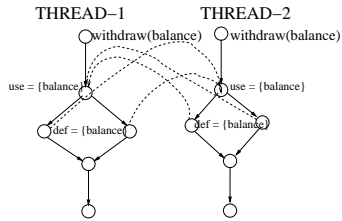


Figure 1. Conflict graph

the variable `balance`. Figure 1 shows the conflict graph. The solid arrows reflect control flow within each thread. The dashed lines (*conflict-edges*) that go across graphs show the potential conflicts between a *definition* in one graph to a *use* of the shared variable `balance` in the other graph. In general, this needs to be done for all shared attributes defined in the object.

The interaction of synchronized blocks with other synchronized blocks or methods requires the construction of conflict edges based on the definition and use of any variable that is locked before the synchronized block. In a general multi-graph, there can be several interactions between the constituting graphs based on whether (1) the region is protected or not, and (2) the variable is being defined or used. As such there can be 16 interactions for 2 threads using one shared variable as shown in Table 1.

Table 1. Interaction between protected/unprotected definition/use

	Protected def	Protected use	Unprotected def	Unprotected use
Protected def	OK	OK	Problem	Problem
Protected use	OK	OK	Problem	OK
Unprotected def	Problem	Problem	Problem	Problem
Unprotected use	Problem	OK	Problem	OK

The example in the graph in Figure 1 shows the interaction based on a *protected-def* and an *unprotected-use*. The graph can be made more detailed by showing separately the computational-uses and predicate-uses of a variable. Construction of these graphs helps us in identifying which variables are likely to have erroneous states if the synchronization has not been performed correctly. We can mutate these variables by applying traditional mutation operators if the variable types are primitive, or mutators described in Bieman et al. [1] for object references.

A mutation engine [3] is used to generate the mutants, drive the tests, record the results and compare the outputs. The mutation engine takes in the program and several configuration items as specifications of the: (1) approach to be used, (2) number of threads to be executed, (3) sequences of methods to be executed in each thread, (4) states and outputs that need to be captured, and (5) test inputs. Fault seeding is only one step in the process of measuring testability. Tech-

niques are needed to monitor the state of the objects and this definitely requires the use of probes to monitor intermediate states, which is a problem. If one is only interested in the end result, probes are not necessary. One still needs to identify a set of observable entities (standard output, standard error messages, changes in the environment — files, etc.).

4. Future Work

This work could become the basis for further research in this area. We considered faults related to race conditions that occur only because of improper (or inadequate) use of synchronization. There are other issues in concurrent programming (e.g. deadlocks) and the use of methods, such as `wait()`, `wait(timeout)`, `notify()` and `notifyAll()`. Scalability experiments need to be performed to investigate the effects of the (1) number of shared variables, (2) number of concurrent threads, and (3) combinations of concurrent methods. Empirical studies need to be performed to assess the effectiveness and utility of using the mutation approach. Since hardware is relatively cheap nowadays, techniques can be developed for distributing the execution of mutants on clusters of workstations to reduce the time of execution. Combined with efforts in applying run-time mutation to objects, this method can be used for real-world applications. The current work can be extended for exploring problems such as deadlock, in addition to the race conditions that were investigated here. The approach can be extended for distributed and real-time systems. The ideas can also be applied to other object-oriented programming languages.

References

- [1] J. Bieman, S. Ghosh, and R. Alexander. “A Technique for Mutation of Java Objects”. In *Proc. of ASE 2001*, San Diego, USA, November 2001.
- [2] R. A. DeMillo and A. J. Offutt. Constraint-based Automatic Test Data Generation. *IEEE Trans. on Software Engr.*, 17(9):900–910, Sep. 1991.
- [3] S. Ghosh. Towards Measurement of Testability of Concurrent Object-Oriented Programs Using Fault Insertion: A Preliminary Investigation. To appear in *2nd IEEE International Workshop on Source Code Analysis and Manipulation*, Montreal, October 2002.
- [4] D. Lea. *Concurrent Programming in Java Second Edition Design Principles and Patterns*. The Java Series. Addison Wesley, USA, November 2000.
- [5] J. M. Voas and G. McGraw. *Software Fault Injection: Innoculating Programs Against Errors*. John Wiley and Sons, January 1998.
- [6] C.-S. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path Coverage for Parallel Programs. In *Proc. of ACM SIGSOFT ISSA*, pp. 153–162, Clearwater Beach, Florida, USA, March 2-5 1998.