Towards Measurement of Testability of Concurrent Object-Oriented Programs using Fault Insertion: A Preliminary Investigation

Sudipto Ghosh
Computer Science Department
Colorado State University
Fort Collins CO 80523, USA
+1 970 491 4608
ghosh@cs.colostate.edu

Abstract

There is a lack of methods and techniques for measuring testability of concurrent object-oriented programs. Current theory and practice for testing sequential programs do not usually apply to concurrent systems. An approach towards measuring the testability of concurrent Java programs is proposed in this paper. The key idea is to take a program and insert faults that are related to the concurrency aspects. The approach is based on two methods: (1) mutation of the program based on keywords, and (2) creation of conflict graphs based on static analysis of the code.

Keywords: *Concurrent programs, faults, Java, mutation analysis, software testing, test adequacy, testability.*

1. Introduction

Software testability [26, 27] 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. Concurrent programs present a challenge to the measurement of software testability. Current theory and practice for testing sequential programs do not usually apply to concurrent programs. Testers of concurrent programs are faced with the problem of non-determinism and non-reproducibility of test results. Race conditions and deadlocks are some of the common problems that need to be detected, but the observation of both is dependent on timing and order of execution. Adding test probes changes the timing characteristics.

We propose an approach that can be used to measure testability of concurrent Java programs. Our approach relies on the insertion of faults into the program without requiring the presence of probes. The faults are selected in two ways: (1) mutation of keywords related to concurrency, and (2) creation of conflict graphs based on definition-use pairs across threads of execution. The location of the keywords can be identified and conflict graphs created by a static analysis of code.

The measurement of testability is closely associated with the development of test adequacy criteria. Test adequacy assessment is the evaluation of how thorough the testing was, and is an indicator of the goodness of the test sets. An adequacy criterion is defined as a predicate that defines what properties of a program must be exercised to constitute a thorough test [14]. The term *coverage domain* denotes a set of program related entities that are checked and counted for measuring coverage. Within a program these entities include functions, statements, decisions, and definition-use pairs. Test coverage is measured with respect to a particular coverage domain and reflects *how much* of that domain has been executed and tested.

Traditionally, software testability measurement approaches require fault seeding into programs. Inserted faults represent plausible errors. A related approach is mutation analysis [9, 10, 20] and it 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. Researchers have advocated the use of syntactic modifications that are determined by a set of *mutation operators*. This set is determined by the language of the program being tested and the mutation system used. Mutation operators are created with one of two goals: (1) to induce simple syntax changes based on errors that programmers typically make (e.g. use a wrong variable name), and (2) to force common testing goals (e.g. execute each branch). Mutation techniques usually apply operators to constructs (variables, constants, operators and statements) in program code. These techniques cannot be easily extended to object-oriented (OO) programs because the variables usually represent objects (actually object references). Some OO mutation approaches use mutation operators that operate at code level to mutate aspects of attribute and method visibility, and inheritance. Bieman et al. [2] proposed run-time mutation of objects created from classes in the Java API library.

Several languages (e.g. C) do not provide support for concurrency in the language itself. Concurrency and synchronization are achieved using libraries that provide constructs like threads, semaphores and mutexes. These constructs can be treated as variables and usual mutation operators may be applied to them. Languages like Java provide synchronization as part of the language (e.g. using the keyword `synchronized`). We are not aware of any research in the application of mutation testing that is targeted to concurrency-related aspects of object-oriented programs. In this paper we propose a new approach to the generation of mutants based on program dependency graphs of concurrent Java programs. Preliminary evaluation of our technique seems to indicate that the approach can be used to systematically seed faults with the goal of measuring testability.

The remainder of this paper is organized as follows. We summarize background material on testability, mutation analysis and testing of concurrent programs in Section 2. We describe our approach and illustrate it with an example in Section 3. In Section 4 we present our conclusions and outline directions for future work.

2. Background

Voas et al. [26, 27] define software testability 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. Testability analysis can show where faults can hide from testing. Voas et al.'s definition differs from other definitions that have been used earlier which usually "discuss the ease with which some input selection criteria can be satisfied during testing." In our work we will adopt Voas et al.'s definition.

Voas developed a technique called PIE analysis [25] (for *propagation*, *infection* and *analysis*). PIE analysis is a "dynamic technique for statistically estimating three program characteristics that affect a program's computational behavior: (1) the probability that a particular section of a program is executed, (2) the probability that the particular section affects the data state, and (3) the probability that a data state produced by that section has an effect on program output."

2.1. Testing Concurrent Programs

Testing for correctness of concurrency properties cannot usually be done by regular sequential program testing techniques. Netzer and Miller [21] propose techniques that would validate the data races detected by previous methods that operate by instrumenting the program so that information about its execution is recorded. State space analyses for deadlock detection have been described by Young [29], Corbett [5, 6] and Godefroid [13]. Tai et al. [3] perform deterministic execution of concurrent software by selecting paths of a concurrent program according to a structure-based criterion. A reachability graph of P is derived and paths selected from the graph. Inputs are derived for each selected path and deterministic executions are forced according to the selected paths.

Chung et al. [4] report on using message sequence charts. This is a specification-based approach and the effectiveness is demonstrated using a telephone switching system example. Dwyer and Clarke [11] propose polynomial-time data flow algorithms that support reasoning about properties of concurrent programs. Developers specify properties as a pattern of selected program events and verify that all or no program executions satisfy the given property. Taylor et al. [24] present structural testing criteria for concurrent programs. The criteria were defined with respect to *concurrency states* of a program. Each concurrency state represents the next synchronization-related activity to occur in each of a system's tasks. Based on these state-spaces, several coverage criteria were defined. The *probe-effect* arising from the insertion of monitoring code changes the runtime behavior of the system.

DU-path based approaches (e.g. [11, 28]) use a graph representation of the concurrent processes (or threads). These graphs are variously called *PPFG* (Parallel Program Flow Graph) or *TFG* (Trace Flow Graph). We have used a similar graph representation for determining (1) what mutant operators to apply, and (2) where to apply them.

2.2. Mutation Testing

Mutation testing relies on the *Competent Programmer Hypothesis* and the *Coupling Effect*. The *Competent Programmer Hypothesis* states that programmers are generally competent and produce a program close to a correct program. A correct program can be constructed from an incorrect program by making changes that are composed of minor alterations. The *Coupling Effect* states that test cases that distinguish programs with minor differences from each other are so sensitive that they can distinguish programs with more complex differences. The competent programmer hypothesis and the coupling effect imply that small changes in programs are adequate to help reveal complex errors.

An example of a programmer error in a C program is typing $x > y$ instead of the intended $x >= y$. Thus, one may define a mutation operator that takes an expression containing $>=$ and replaces it with $>$. The goal for the tester is to create a test case that can detect a difference in the outputs of the program containing $>=$ and the mutant containing $>$. Other examples of mutation operators include (1) replace a binary operator by another, (2) replace a variable by another variable used in the program, and (3) replace a constant by another constant.

Let P be the program under test and P_c be one correct version of P. If P is correct, P and P_c are the same. $\mathcal{T}$ is the set of tests used to test P. Let the input domain of P be denoted by $\mathcal{D}$. Mutation testing relies on a set of faults $\mathcal{F}$. Each fault f in $\mathcal{F}$ is introduced in P one by one. Introduction of a fault into P results in a program M that is slightly different from P. The program M is called a *mutant* of P. The application of all the faults in $\mathcal{F}$ one by one into P produces a set of mutants $\mathcal{M}$. Elements of $\mathcal{F}$ are known as *mutation operators*. When a mutant M is executed against a test case t in $\mathcal{T}$ and the behavior of M is different from that of P, the mutant M is said to be *killed* by t. A tester is expected to kill each mutant in $\mathcal{M}$ with at least one test case t. In case a mutant cannot be killed, the tester needs to show that $M \equiv P$ or update $\mathcal{T}$ by adding a test case $t' \notin \mathcal{T}$ that kills M. A mutant that is not killed during testing is said to be *live*. The adequacy of $\mathcal{T}$ is determined by the ratio of the number of mutants killed to the number of non-equivalent mutants in $\mathcal{M}$. This ratio is also called the mutation score. $\mathcal{T}$ is considered adequate if the mutation score is 1.

Mutation operators were defined for procedural programming languages such as Fortran [23] and C [1, 7, 8]. Mutation operators defined for C or Fortran may also apply to OO languages (e.g Ada and Java) with small changes in syntax. However, Ada and Java have several additional features that arise out of the OO programming paradigm. Examples of early work in OO mutation analysis includes that by Offutt et al. [22], who applied mutation analysis to Ada programs and addressed some of the object-oriented features, but were limited to properties within a class and did not address inheritance.

Kim et al. [17, 18] used Hazard and Operability Studies Analysis (HAZOP) to define mutation operators for Java. HAZOP was applied to the Java syntax definition to identify deviations of Java language constructs. The analysis provided a list of Java mutation operators and helped create a database of flaws for Java programs. The operators apply to types and variables, modifiers, class and interface declarations, blocks and expressions. Specifically, class mutation targets faults related to the OO-specific features of Java: (1) class declarations and references, (2) single inheritance, (3) information hiding and (4) polymorphism. Kim et al.'s approach did not address the issue of integration testing, nor did their operators always result in mutants that compile. More importantly, they did not address synchronization aspects of the Java language. Kim et al. [16] also used their mutation method to compare the effectiveness of object-oriented testing strategies. Bieman et al. [2] proposed a runtime approach for mutating Java objects, but did not address the problem of concurrency. Jackson and Woodward [15] describe a parallel firm mutation analysis of Java programs. While the mutations are executed in parallel (i.e. the mutation technique uses concurrent threads), the mutation operators themselves do not address concurrency in Java programs.

Delamaro et al. [8] first described the technique of *interface mutation* for the integration testing of C programs. The underlying idea is to create mutants by inducing simple changes only in the entities belonging to the interface between modules or sub-systems. The interface mutation approach (1) restricts the mutation operators to model only integration errors, (2) tests only the connections between two modules, a pair at a time, and (3) applies the integration mutation operators only to module interfaces such as function calls, parameters or global variables. Ghosh and Mathur [12] applied interface mutation analysis to distributed object systems.

3. Approach

In this section we examine the Java concurrency model and explain how mutation analysis may be applied to concurrent Java programs. We first describe the fault model and then show two approaches for creating mutants. Then we describe the features of a mutation engine that may be used to automate the analysis. We also discuss the implications of using mutation analysis on concurrent programs.

Concurrent programs need to be concerned about safety and liveness. "In live systems, every activity eventually progresses toward completion; every invoked method eventually executes." [19] Although programs may fail to make progress for reasons such as locking, waiting, input, CPU contention or failure, there may be more permanent lack of progress because of deadlock, missed signals, nested monitor lockouts, livelock, starvation, resource exhaustion or distributed failure [19]. Although there may be programming errors that are similar to errors in sequential programs, the focus of this paper is on errors related to concurrency.

Object-oriented concurrent programming is similar to the concurrent programming done in procedural languages such as C, but usually differ due to encapsulation and OO features. Moreover, languages like Java provide concurrency support in the programming language itself, thereby making it safer to program because certain classes of errors are ruled out (e.g. forgetting to release locks upon exit).

Java provides two syntactic forms for synchronization. These forms which are based on the `synchronized` keyword, can be used for blocks or methods. Block synchronization takes an argument of which 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 [19] for details).

Programmers may make errors in the use of locking. In general, safety and liveness go in opposite directions, i.e. more liveness may imply less safety and vice versa. In this paper, our focus is more on the correctness (or safety) aspects rather than on the liveness aspects. In other words, even though programmers may perform more locking than necessary, thereby leading to liveness problems, we are not concerned with such cases. Instead we focus on those cases where access to shared data needs to be protected, but is not. We will illustrate the mutation analysis approach for addressing such problems. One can anticipate problems related to deadlock, both on single machines, and on multiple machines (distributed deadlock). Such problems are not included in the scope of our paper.

Our fault model assumes that programmers may not use the `synchronized` construct where they should. For example, there may be two methods `m1` and `m2` that access and modify the same variable `v` but `m1` is synchronized and `m2` is not. This causes problems when thread t_1 is executing `m1` and thread t_2 (which is not blocked) is executing `m2`. Another example is when both `m1` and `m2` and not synchronized. Similar examples may occur with the use of static fields as well. For synchronized blocks, the error may be to omit the synchronization directive. All these errors may arise because of poor design, carelessness, or lack of knowledge about synchronization rules in Java.

3.1. Simplistic approach

The first approach does not require the creation of a graph representation of contentions in threads. We define a pair of operators `RSYNCM` and `RSYNCB` related to the removal of the keyword `synchronized`.

- `RSYNCM`: This operator removes the occurrence of the `synchronized` keyword from the text of a method and creates a mutant without the keyword. This results in a mutant that does not lock the method. The goal of the tester is to develop test cases that can detect differences in the output of execution of the original program and the mutant. The following are some important issues to consider:

  1. It may be necessary to apply different sequences of methods to increase the chances of creating differences in outputs along with differences in the test cases themselves. This brings up the issue of the determination of equivalent mutants. For example, if `m1` is synchronized and `m2` is not, then two threads that concurrently run only `m1` will not detect any difference. Thus the mutants will be equivalent. However, threads that run `m1()` and `m2` in parallel (or both run `m2`) will be more likely to detect differences. Using mutation analysis requires the tester to detect differences in the outputs, and therefore, forces different sequences to be used.
  2. 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. 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. This may make it difficult to see differences in output, for example, if there is no interleaving at all.

- `RSYNCB`: This is similar to the `RSYNCM` operator, except that it is applied to a block. The same issues are valid here as well.

The problem of our simplistic approach is that the tester does not have control over timing. It may also be difficult to control the number of threads, for example, in Java RMI, the implementer of the remote object code has no control over the number of threads generated by the underlying RMI system. In the next subsection, we propose an approach that addresses this issue and provides greater control.

Similar to the operators that remove the `synchronized` keyword, we may define new ones that add the `synchronized` keyword to a method or block that does not have it and then observe the effect on the liveness (less concurrency, or deadlocks). However, this approach is beyond the scope of the paper.

3.2. Program conflict graph-based approach

Given the source code of the program, we can construct a conflict-graph (similar to control-flow graphs described in the literature on testing concurrent software, such as [11]) 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.

3.2.1 Method and method interaction

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

```
public synchronized void withdraw(int amt) {
    if (balance > (amt+TRANSACTION_FEE) {
        balance = balance - amt - TRANSACTION_FEE;
    } else {
        balance = balance - TRANSACTION_FEE;
    }
    return;
}
```

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.

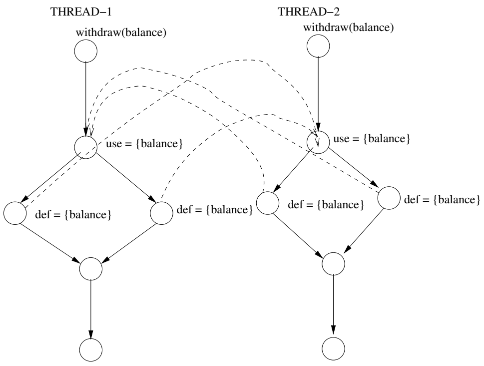

Figure 1. Conflict graph - I.

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 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.

3.2.2 Block and block/method interaction

The interaction of synchronized blocks with other synchronized blocks or methods is more complex. The synchronized block construct allows the programmer to lock an object which is usually the same object on which the method is being invoked (this), an attribute of the same object, or some other object. In such cases, the construction of the conflict edges is based on the definition and use of any variable that is locked before the synchronized block.

3.2.3 More on conflict graphs

In general, conflict graphs take the form shown in Figure 2 which shows the interaction between two threads that may be executing some methods f and g. The region inside the rectangle shows that the variable v is protected (such as by

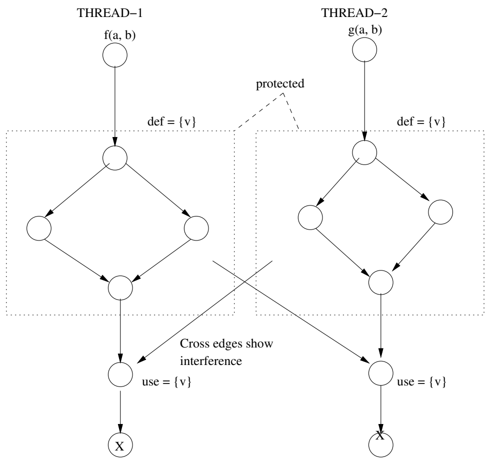

Figure 2. Conflict graph - II.

using the synchronized keyword in Java). There can be interactions between the two graphs that constitute the multi-graph, 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.

The example in the graph shows the interaction based on a *protected-def* and an *unprotected-use*. In Java, if Method1 and Method2 are methods in the same class such that Method1 is synchronized and Method2 is not, then both methods could execute simultaneously on the same piece of data. The graph can be made more detailed by showing separately the computational-uses and predicate-uses of a variable. These activities can be performed by static analysis of the program.

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 to apply plausible faults to the program. Operators can be applied to increment or decrement values if the type is primitive. Otherwise, for objects we can use mutators as defined in Bieman et al. [2].

3.3. Mutation engine

The objective of the mutation engine is 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.

For the first approach, the mutation engine generates mutants by removing the synchronization directives. The mu-

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

tants are generated as code which is compiled and executed by the driver. Figure 3 shows how the mutation engine works in this case. The engine invokes the test driver that invokes methods on the mutant code as specified by the *method sequence specs*. The state and output of the mutants are captured by the state recorder based on the specifications of the *state specs*. This output is stored in the form of *results* that are compared with the *Oracle* using a *comparator*. The test outcome is similarly obtained for all the other mutants.

The second approach that requires building a conflict graph is almost the same, except that the mutant code generation is not part of the process. Generation of conflict-graphs can be done using a static analysis of the code. However, the actual mutation is a non-trivial task. For sequential programs, one can generate code that has mutation operators applied to variables. We could take the same approach for concurrent programs as well, but then all the threads would operate on the same mutant code. The goal of our approach is to have one thread working on one version and the other thread working on the mutant version to simulate flaws in the synchronization of definition and use of variables. This is not straightforward and will require support from the underlying virtual machine. Work is in progress to work on an alternative design to enable this aspect.

3.4. Discussion

The example shown has been made simple to demonstrate how the approach works. In reality, there may be more variables that are shared, and also some that are not shared. If one used traditional mutation, all the variables would have to be mutated, and this would lead to a large number of mutants. Our approach reduces the number of mutants to correspond to only those variables that have potential conflicts.

The conflict-graph construction described here for dual-threaded programs only looks at a pair of methods at a time. In our example, both threads were executing the same methods. However, there is nothing in our approach that prevents us from investigating two threads that execute two different methods at the same time. Moreover, depending on the timing characteristics, the first thread t_1 originally executing method `m1` may start executing a different method `m2` while the second thread t_2 is still executing `m3`. For a more complete analysis, one needs to look at all possible combinations of methods that can execute on the two threads and interact.

While our approach has been demonstrated for a dual-threaded program, the analysis becomes much more complicated if there is a larger number of threads. We need to empirically evaluate the scalability of the method with respect to the degree of concurrency. The number of mutants will definitely grow *quickly* with the increase in the number of threads. However, it is likely (though no experimental results are currently available) that if one can show a high degree of testability with a small number of threads, the effects will be more pronounced for a larger number of threads. The rationale behind this kind of reasoning is that a larger number of threads makes it all the more likely to reveal problems. It is usually more difficult to find concurrency flaws with fewer threads. This observation implies that one does not need to use a large number of threads to measure testability.

In our work we have 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). Java allows programmers to use methods, such as `wait()`, `wait(timeout)`, `notify()` and `notifyAll()`. Improper use of these methods can cause liveness problems (livelock and deadlock).

4. Conclusions and Future Work

We demonstrated two methods for inserting concurrency-related faults into multi-threaded Java programs. The methods give us plausible faults to insert for measuring testability of the programs. The faults and the locations are identified by a static analysis of the program code and can be completely automated. Our mutation engine can be used to generate the mutant programs and drive the test process. We are working on the design and implementation of the thread mutation part in the test driver for the second approach.

Fault seeding is only one step in the process of measuring testability. Techniques are needed to monitor the state of the objects and this definitely requires the use of probes

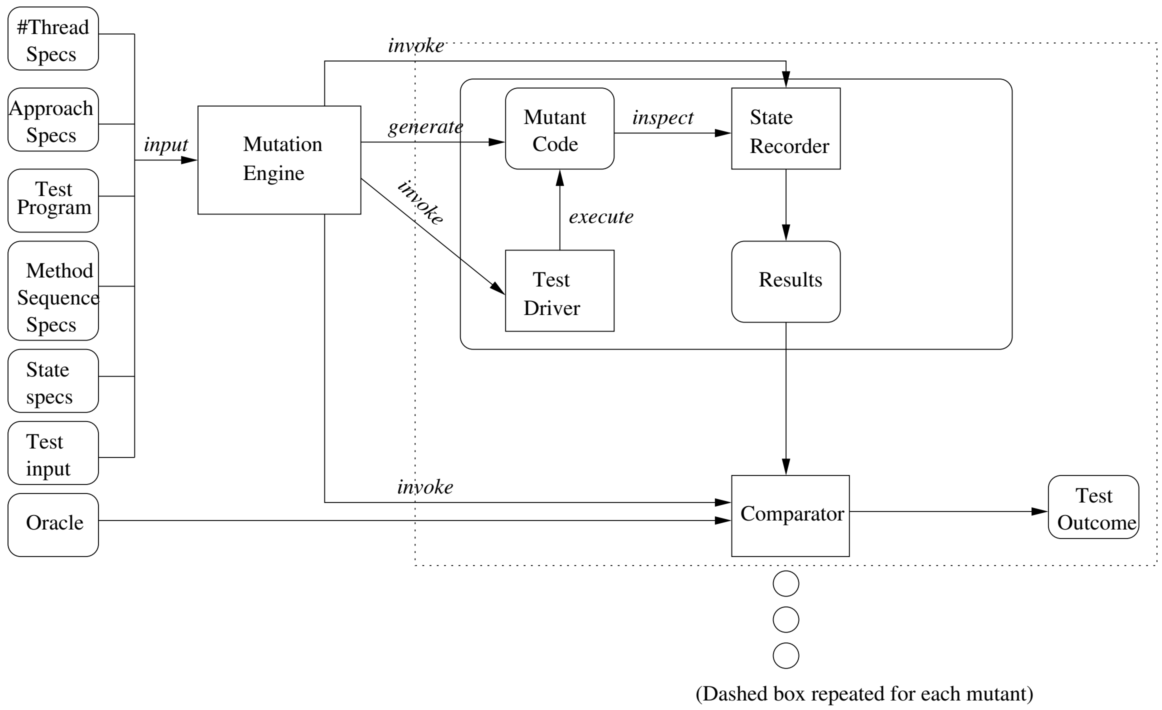

Figure 3. Mutation Engine - I.

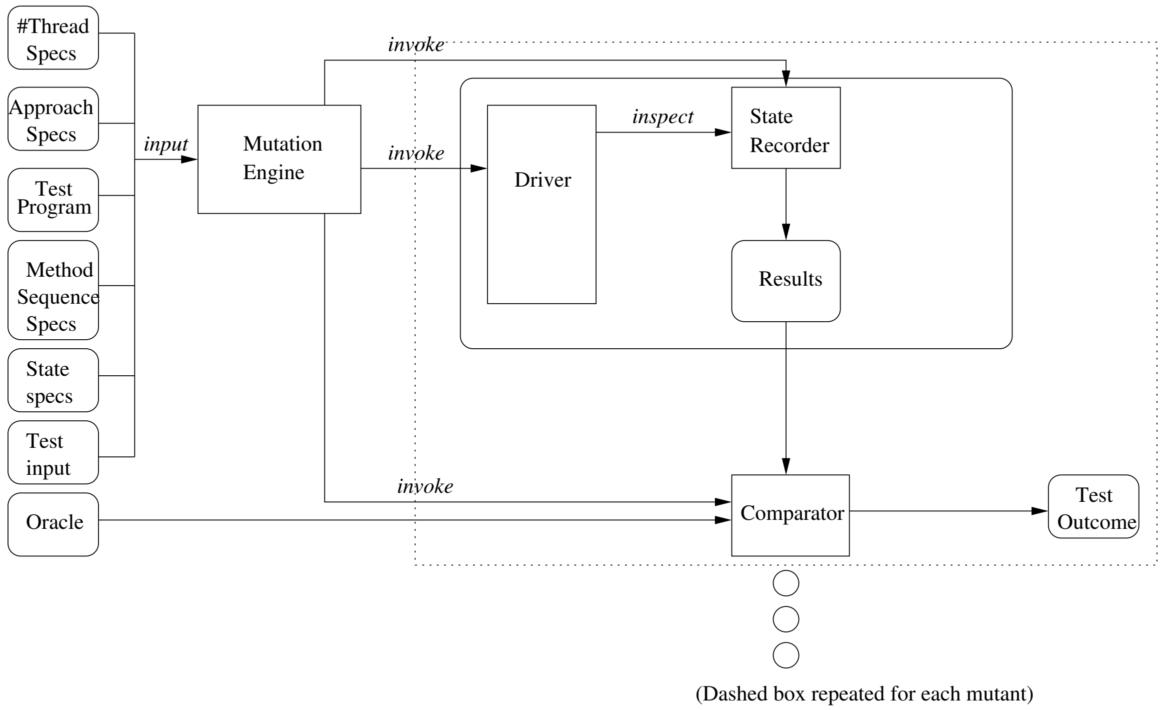

Figure 4. Mutation Engine - II.

to monitor intermediate states, which is a problem. If one is only interested in the end result, probes are not necessary. Of course, one still needs to identify a set of observable entities (standard output, standard error messages, changes in the environment — files, etc.).

This paper could become the basis for further research in this area. Empirical studies can be performed to evaluate (1) the scalability of the proposed method with respect to the degree of concurrency, (2) the ideal number of concurrent threads to run to achieve a certain degree of confidence in the effort, and (3) 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] H. Agrawal, R. DeMillo, R. Hathaway, W. M. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, and E. Spafford. Design of Mutant Operators for the C Programming Language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, March 1989.

[2] J. Bieman, S. Ghosh, and R. Alexander. "A Technique for Mutation of Java Objects". In *Proceedings of Automated Software Engineering (ASE 2001)*, San Diego, USA, November 2001.

[3] R. Carver and K. C. Tai. Deterministic Execution Testing of Concurrent Ada Programs. In *Annual International Conference on Ada, Conference proceedings on Ada technology in context: application, development, and deployment*, pages 528–544, Pittsburgh, Pennsylvania USA, October 22-26 1989.

[4] I. S. Chung, H. S. Kim, H. S. Bae, Y. R. Kwon, and B. S. Lee. Testing of Concurrent Programs Based on Message Sequence Charts. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, Los Angeles, California USA, May 17-18 1999.

[5] J. C. Corbett. An Empirical Evaluation of Three Methods for Deadlock Analysis of Ada Tasking Programs. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, pages 204–215, Seattle, Washington, USA, August 17-19 1994.

[6] J. C. Corbett. Evaluating Deadlock Detection Methods for Concurrent Software. *IEEE Transactions on Software Engineering*, 22(3):1–22, March 1996.

[7] M. E. Delamaro, J. C. Maldonado, M. Jino, and M. L. Chaim. PROTEUM: Uma ferramenta de teste baseada na analise de mutantes (proteum: A test tool based on mutation analysis). In *Software Tools Proceedings of the VII Brazilian Symposium on Software Engineering*, October 1993.

[8] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Integration Testing Using Interface Mutation. In *Proceedings of International Symposium on Software Reliability Engineering (ISSRE '96)*, pages 112–121, April 1996.

[9] R. A. DeMillo, D. S. Guindi, K. N. King, and W. M. McCracken. An Overview of the Mothra Software Testing Environment. Technical Report SERC-TR-3-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, August 1987.

[10] R. A. DeMillo and A. J. Offutt. Constraint-based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[11] M. B. Dwyer and L. A. Clarke. Data Flow Analysis for Verifying Properties of Concurrent Programs. In *Proceedings of the second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 62–75, New Orleans, Louisiana USA, December 6-9 1994.

[12] S. Ghosh and A. P. Mathur. "Interface Mutation". *Journal of Software Testing, Verification and Reliability*, 11(4):227–247, December 2001.

[13] P. Godefroid. Verisoft: A Tool for the Automatic Analysis of Concurrent Reactive Software. In *Proceedings of the 9th Conference on Computer Aided Verification, Lecture Notes in Computer Science, vol. 1254*, pages 476–479, Haifa, June 1997.

[14] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, 1975.

[15] D. Jackson and M. Woodward. "Parallel Firm Mutation of Java Programs". In *Proceedings of Mutation 2000, Kluwer Academic Publishers*, pages 55–61, San Jose, USA, October 2000.

[16] S. Kim, J. Clark, and J. McDermid. "Investigating the Effectiveness of Object-Oriented Strategies with the Mutation Method". *Journal of Software Testing, Verification and Reliability*, 11(3):207–225, December 2001.

[17] S. Kim, J. A. Clark, and J. A. McDermid. "The Rigorous Generation of Java Mutation Operators Using HAZOP". In *Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications (ICSSEA'99)*, Paris, France, December 1999.

[18] S. Kim, J. A. Clark, and J. A. McDermid. "Class Mutation: Mutation Testing for Object Oriented Programs". In *Proceedings of the FMES 2000*, 2000.

[19] D. Lea. *Concurrent Programming in Java Second Edition Design Principles and Patterns*. The Java Series. Addison Wesley, USA, November 2000.

[20] A. P. Mathur and W. E. Wong. A Theoretical Comparison Between Mutation and Data Flow Based Criteria. In *Proceedings 22nd Annual ACM Computer Science Conference on scaling up: meeting the challenge of complexity in real-world computing applications*, pages 38–45, Phoenix, Arizona, USA, March 8-10 1994.

[21] R. H. B. Netzer and B. P. Miller. Improving the Accuracy of Data Race Detection. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 133–144, Williamsburg, Virginia USA, April 21-24 1991.

[22] A. J. Offutt, J. Voas, and J. Payne. Mutation Operators for Ada. Technical Report ISSE-TR-96-09, Information and Software Systems Engineering, George Mason University, Fairfax, Virginia, March 1996.

[23] R. A. DeMillo et al. An Extended Overview of the MOTHRA Testing Environment. In *Workshop of Software Testing, Verification and Analysis*, July 1988.

[24] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural Testing of Concurrent Programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, March 1992.

[25] J. M. Voas. PIE: A Dynamic Failure-based Technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, August 1992.

[26] J. M. Voas and G. McGraw. *Software Fault Injection: Innoculating Programs Against Errors*. John Wiley and Sons, January 1998.

[27] J. M. Voas and K. W. Miller. "Software Testability: The New Verification". *IEEE Software*, pages 17–28, May 1995.

[28] C.-S. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path Coverage for Parallel Programs. In *Proceedings of ACM SIGSOFT international symposium on Software testing and analysis*, pages 153–162, Clearwater Beach, Florida, USA, March 2-5 1998.

[29] M. Young. State-Space Analysis as an Aid to Testing. In *Proceedings of the 1994 International Symposium on Software testing and Analysis*, page 203, Seattle, Washington USA, August 17-19 1994.