Comparative Assessment of Testing and Model Checking Using Program Mutation

Jeremy S. Bradbury
Faculty of Science
University of Ontario Institute of Technology
Oshawa, Ontario, Canada
Jeremy.Bradbury@uoit.ca

James R. Cordy, Juergen Dingel[2]
School of Computing
Queen's University
Kingston, Ontario, Canada
{cordy, dingel}@cs.queensu.ca

Abstract

Developing correct concurrent code is more difficult than developing correct sequential code. This difficulty is due in part to the many different, possibly unexpected, executions of the program, and leads to the need for special quality assurance techniques for concurrent programs such as randomized testing and state space exploration. In this paper an approach is used that assesses testing and formal analysis tools using metrics to measure the effectiveness and efficiency of each technique at finding concurrency bugs. Using program mutation, the assessment method creates a range of faulty versions of a program and then evaluates the ability of various testing and formal analysis tools to detect these faults. The approach is implemented and automated in an experimental mutation analysis framework (ExMAn) which allows results to be more easily reproducible. To demonstrate the approach, we present the results of a comparison of testing using the IBM tool ConTest and model checking using the NASA tool Java PathFinder (JPF).

1. Introduction

In order to fully exploit recent hardware advances such as multi-core processors, software needs to be concurrent. In the past, advances in single processors have lead to free speed-up of sequential programs which will no longer occur with multi-core technologies. Many imperative programming languages like Java, which are often used in the development of sequential programs, can also be used for handling concurrency. For example, Java provides a number of synchronization events (wait, notifyAll) for the development of concurrent programs. The synchronization events in Java can affect the scheduling of threads and access to variables in the shared state [7]. The interleaving space of a concurrent Java program consists of all possible thread schedules [12].

The development of concurrent software offers a set of challenges not present in the development of sequential code. For example, in software with multiple threads deadlock and race conditions can occur. Furthermore, a fault that leads to a deadlock or race condition may only occur in a very small number of execution interleavings meaning it is extremely difficult to detect it prior to software deployment.

Our work focuses primarily on better understanding fault detection techniques for concurrent software. Many approaches to ensuring concurrent Java source code is correct have been proposed including: concurrency testing (e.g., ConTest [11]), model checking (e.g., Java PathFinder [15, 22, 1], Bandera/Bogor [19]), dynamic analysis (e.g., ConAn [17]) and static analysis (e.g., FindBug [16]). The goal of this paper is to compare two existing fault detection techniques – namely concurrency testing with the IBM tool ConTest [11] and model checking with NASA's Java PathFinder (JPF) [15, 22, 1]. With respect to these two tools we ask the following questions:

Which technique is more effective?
Which is more efficient?

Effectiveness refers to the ability of each tool to detect concurrency faults and *efficiency* refers to how quickly each tool is capable of finding faults. Our interest in exploring the relationship between testing and model checking tools is motivated by a need for improved quality assurance techniques for concurrent industrial source code. Testing has been an effective industrial technique for fault detection of sequential programs while model checking tools offer the potential to substantially aid in the debugging of concurrent programs.

[1]this work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

[2]from Jan. to Aug. 2007 on leave to Software Systems Engineering Institute, Braunschweig University of Technology, Braunschweig, Germany.

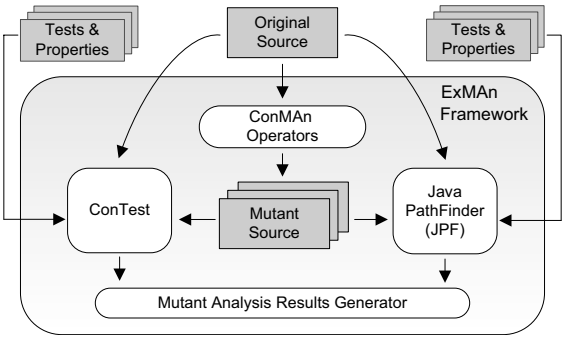

Figure 1. Experimental mutation analysis using the *ExMAn* framework and the *ConMAn* operators

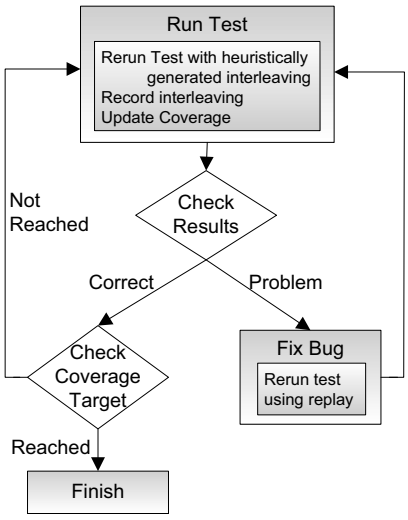

Figure 2. Testing with ConTest [11]

We conducted a controlled experiment to compare ConTest and JPF. Our research approach for comparing testing and model checking is a generalization of mutation testing. Program mutation is traditionally used to evaluate the effectiveness of test suites. It provides a comparative technique for assessing and improving multiple test suites. A number of empirical studies (e.g., [2, 8]) have relied on using mutants as a proxy for real faults during the experimental process. The use of program mutation as a proxy has been well researched. Jeff Offutt studied the coupling effect of simple mutant faults with more complicated faults [18]. Andrews, Briand and Labiche have studied the relationship between mutant faults and real faults with respect to sequential source code [2]. The previous studies on mutation testing establish a firm basis for the use of mutation in empirical research.

Although mutation as a comparative technique has been used primarily within the testing community, it does have application in the broader area of fault detection techniques. Our work is based on the idea that mutation can be used to assess testing, model checking, static analysis and dynamic analysis. In previous work we have detailed our Experimental Mutation Analysis (*ExMAn*) framework which implements our research approach [3]. We have also defined a set of Concurrency Mutation Analysis (*ConMAn*) operators to support program mutation with concurrent Java [4]. We use *ExMAn* in combination with *ConMAn* to automate the research methods used in our controlled experiment (see Figure 1).

In Section 2 we overview concurrency testing with ConTest and model checking with JPF. We outline our experimental goals in Section 3, our experimental setup in Section 4 and our experimental procedure in Section 5. Our experimental results and outcomes are given in Section 6 and threats to validity are discussed in Section 7. Finally, we review related work and present our conclusions in Sections 8 and 9.

2 Background

A variety of quality assurance techniques are used to detect faults in concurrent programs. We begin with a brief background on two of these techniques – testing and model checking.

2.1. Testing

Conventional testing of sequential programs usually involves developing a set of test cases that provide a certain kind of code coverage (e.g., path coverage). These tests are executed on the code to detect possible faults and failures. If a fault is detected in sequential code for a given test case then we can rerun the test case to demonstrate the fault and reuse the test case on a new version of the software to ensure that the fault no longer occurs.

Due to the non-determinism of the execution of concurrent source code and the high number of possible interleavings, concurrency testing can not rely on coverage metrics alone to guarantee the quality of the source code. In addition to ensuring that all code is covered we must also provide some probabilistic confidence that faults that manifest themselves in only a few of the interleavings are found. For example, since a race condition or deadlock may only occur in a small subset of the possible interleaving space, the more interleavings we test the higher our confidence that the fault that causes the race condition or deadlock will be found [21].

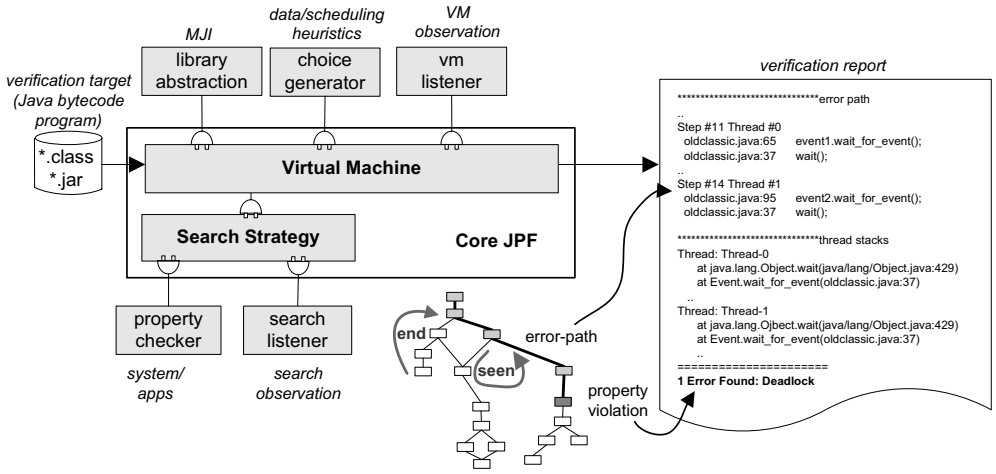

Figure 3. Model Checking with Java PathFinder [1]

Although conventional testing can find faults in concurrent source code it may not be capable of finding all faults. ConTest is an IBM concurrency testing tool developed specifically to enhance the ability of testing to find more faults by exploring the interleaving space of the program [11]. ConTest works with concurrent Java and uses a randomized scheduler and heuristics to increase the confidence that interleavings with a high-risk of faults are explored (see Figure 2). Randomized scheduling is obtained by automatically and systematically inserting delays into Java bytecode. For example, one way to delay the execution of a thread is to cause it to sleep for a random amount of time.

2.2. Model Checking

Software model checking is a formal methods approach that typically involves developing a finite state model of a software system and specifying a set of assertions or temporal logic properties that the software system should satisfy. The model checker determines if the model of the software system satisfies the specified properties by conducting an exhaustive state space search. The exhaustive search means that all possible interleavings of the model of a concurrent system are examined and thus provides a high leave of confidence regarding the quality of the software. Although model checking can provide more confidence then testing it typically requires a long time to search the state space.

Traditionally model checkers are used to prove correctness, however model checkers also provide benefits as debuggers. A shift in the focus of techniques like model checking from proofs of correctness to debugging and testing has been advocated by a number of researchers including Rushby [20]. The ability of today's state-of-the-art software model checkers to directly analyze source code and the increase in size of systems that can be analyzed has helped them become a viable option for software debugging. For example, in most model checkers a counter-example is produced if the verification of a property fails. When a counter-example is produced it can be used to locate the error in source code.

Several software model checkers support the analysis of concurrent Java including JPF and the Bandera/Bogor tool set, developed at Kanas State University. In this paper we have chosen to use JPF for our experiment however in the future we also plan to conduct further experiments with Bandera/Bogor.

JPF (see Figure 3) is an example of an explicit state model checker. It uses Java bytecode as an input language and thus eliminates the semantic gap between source and model artifacts. It also uses a special virtual machine to manage the program state. JPF is fairly flexible and can use different algorithms to search the state space. Search algorithms supported by JPF include a depth first search, breadth first search, heuristic searches, and a random search. JPF is also flexible in terms of the kind of properties detected. By default it detects deadlocks and exception violations but the user can also create custom properties such as race condition detection. Upon detecting a property violation JPF will provide both the property that was violated and the error path as output to the user.

3. Experimental Definition

The goal of our controlled experiment is to statistically evaluate ConTest and JPF for concurrent Java using measurements to determine both the effectiveness the efficiency

of each tool at finding faults.

In terms of effectiveness, there are two outcomes that are most likely. One, ConTest and JPF are *complementary*. For example, it may be the case that JPF can find bugs that ConTest can not find and vice versa. Two, ConTest and JPF are *alternatives*. For example, it may be the case that both tools are equally likely to find most of the faults in a concurrent program. In this situation the use of both techniques in combination would provide very little, if any, benefit over either approach in isolation.

In terms of efficiency, there are three possible outcomes that are most likely. One, it might be the case that overall ConTest or JPF is *more efficient*. Two, a *mixed result* is possible where in certain cases ConTest is more efficient and in other cases JPF is more efficient. Three, it may be the case that there is *no statistical difference* between the efficiency of ConTest and JPF.

In order to determine the actual outcome of our controlled experiment we collected 3 measurements: mutant score, ease to kill a kind of mutant and cost to kill a mutant.

To evaluate the effectiveness of ConTest and JPF at detecting (killing) faults we use the mutant score. The mutant score provides a good comparative measurement to quantify the ability of different fault detection techniques at finding mutant faults.

> **mutant score of *t* =** *the percentage of mutants detected (killed) by a technique* t *(e.g., ConTest, JPF)*

To evaluate the effectiveness of ConTest and JPF at detecting a particular kind of fault we measure the ease to kill a kind of mutant. The ease to kill a mutant is a measurement used by Andrews et al. [2]. We use ease to kill to help identify any relationships regarding the kinds of faults that are found by a given tool.

> **ease to kill a kind of mutant by *t* =** *the percentage of mutants of a given kind that are detected (killed) by a technique* t .

To evaluate the efficiency of ConTest and JPF at detecting faults we measure the cost to kill a mutant. We record both the real time and the CPU time required to detect faults and can compare tools based on either time.

> **cost to kill a mutant by *t* =** *the total time to detect (kill) the mutant by a technique* t

4. Experimental Setup

In this section we define the setup of our experiment. The setup involves selecting the approaches under comparison, the example programs used in the experiment, the mutation operators used to generate faults, the quality artifacts used by the approaches under comparison and the experimental environment. As we describe the selection of each part of the experimental setup we will justify our choices by answering important questions that may affect the validity of the experiment.

4.1. Selection of Approaches for Comparison

We have already outlined testing with ConTest in Section 2.1 and model checking with JPF in Section 2.2. Recall that JPF can be customized with different search algorithms which explore the program state space in different ways. We have chosen to use JPF with a depth-first search and non-random scheduling that will exhaustively explore the entire state space of the abstracted program. We have selected this configuration for JPF because it is the default configuration.

Are the approaches or tools applied to the same kinds of applications? Both ConTest and Java PathFinder are intended to be used with concurrent Java applications indicating that they are appropriate tools for comparison.

Do the approaches or tools have similar goals? Both ConTest and JPF can be used for the detection of faults in concurrent applications. However, one difference in terms of the goals of each tool is that ConTest is not intended to automatically detect deadlock faults while Java PathFinder is intended to find deadlocks in addition to other kinds of faults. To allow our testing approach to find deadlock we combine ConTest with the Java Virtual Machine's (JVM) Ctrl-Break handler. The Ctrl-Break handler provides a thread dump of a running program and performs deadlock analysis on the program reporting any deadlocks detected. The combined use of ConTest and the Ctrl-Break handler ensures that both testing and model checking are capable of detecting the same kinds of faults.

4.2. Selection of Example Programs

We selected 4 example programs from the IBM Concurrency Benchmark [13] for our experiment:

- *TicketsOrderSim:* A simulation program in which agents sell airline tickets.
- *LinkedList:* A program that has two threads adding elements to a shared linked list.
- *BufWriter:* A simulation program that contains a number of threads that write to a buffer and one thread that reads from the buffer.
- *AccountProgram:* A banking simulation program where threads are responsible for managing accounts.

Example Program	loc	classes	methods	stmts	synch blocks	synch block stmts	synch mthds	synch mthd stmts	critical regions	critical region stmts
TicketsOrderSim	75	2	3	21	1	6 (28.6%)	0 (0%)	0 (0%)	1	6 (28.6%)
LinkedList	303	5	22	70	2	4 (5.7%)	0 (0%)	0 (0%)	2	4 (5.7%)
BufWriter	213	5	9	55	3	20 (36.4%)	0 (0%)	0 (0%)	3	20 (36.4%)
AccountProgram	145	3	7	40	2	3 (7.5%)	3 (42.9%)	5 (12.5%)	5	8 (20%)

Table 1. Metrics for the example programs used in our experiment

Are the example programs representative of the kinds of programs each approach is intended for? Selecting our example programs from an existing benchmark was our best opportunity to find examples that are representative of concurrent Java applications. However, it is important to note that the programs in the benchmark use synchronized blocks and methods to protect access to shared data (see Table 1) and none of the programs use the new J2SE 5.0 concurrency mechanisms. We had difficulty finding example programs that used these mechanisms and plan to conduct further experiments in the future once these mechanisms become more widely used. Another reason for not including the new concurrency mechanisms is that they are not fully supported in ConTest. Therefore, our example programs are not representative of all concurrent Java programs written with J2SE 5.0 mechanisms like semaphores, built-in thread pools and atomic variables.

Are the example programs developed by an independent source? The example programs in the IBM Benchmark were all developed by independent sources however we did have to make modifications to the programs in order to facilitate their use in our experiment. Modifications were required because all of the programs in the benchmark had existing faults and our mutation-based setup requires *correct* original programs to compare with mutants. Therefore we modified each of the programs by hand to fix existing faults. In fixing the faults we were careful to use the same synchronization techniques used in other parts of the program. In order to assure that the modified programs were *correct* we defined *correctness* as any program that can be executed for a fixed amount of time in ConTest without uncovering a fault and can be model checked in JPF for a fixed amount of time without uncovering a fault.

4.3. Selection of Mutation Operators

In our experiment we decided to use a subset of the *ConMAn* operators (see Table 2) that mutate the Java concurrency mechanisms prior to J2SE 5.0. For examples of the *ConMan* operators see [4].

Are the mutation operators modifying parts of the source code analyzed or tested by the approaches under comparison? The operators are ideal for this comparison because they modify the concurrency parts of the source code tested by ConTest and analyzed by JPF.

Name	Description
MXT	Modify Method-*X* Time (wait(),sleep() and join() method calls)
MSP	Modify Synchronized Block Parameter
RTXC	Remove Thread Method-*X* Call (wait(),sleep(), join(), yield(), notify() and notfiyAll() method calls)
RNA	Replace notfiyAll() with notify()
RJS	Replace join() with sleep()
ASTK	Add static keyword to method
RSTK	Remove static keyword from method
ASK	Add synchronized keyword to method that contains a synchronized block
RSK	Remove synchronized keyword from method
RSB	Remove synchronzied block
RVK	Remove volatile keyword
SHCR	Shift critical region (up and down)
SKCR	Shrink critical region
EXCR	Expand critical region
SPCR	Split critical region

Table 2. A subset of the *ConMAn* operators [4] used in our experiment

Are the mutants generated by the mutant operators detectable using the approaches? The operators are also ideal because they generate mutants that ConTest and JPF are capable of detecting. Our only concern regarding the capability of ConTest and JPF to detect the mutant faults was the ability of ConTest to detect mutants that may cause deadlock. We have alleviated this concern by using ConTest with the JVM's Ctrl-Break handler as described in Section 4.1.

4.4. Selection of Quality Artifacts

In our experiment we use test inputs and assertions. In most cases the programs have preexisting test inputs in the form of a driver class and each have one embedded property regarding the correctness of the program upon termination.

Are the artifacts of any approach more mature or advanced? Do the artifacts of one approach provide an advantage over the artifacts of another approach? In order to ensure that testing with ConTest or model checking with JPF

does not use more mature quality artifacts we have decided to use the same artifacts for both techniques. Specifically, we use fixed test inputs and built-in assertions when testing and model checking. On the one hand, ConTest can detect assertion violations at run-time in addition to using test inputs. On the other hand, JPF can search the state space of a program for assertion violations with respect to specific test inputs.

4.5. Selection of Experimental Environment

The environment used for the experiment was a single user, single processor machine (Pentium 4 3GHz) with 3 GB of memory running the ubuntu Linux operating system.

Are there any factors in the experimental environment that can give one approach an advantage? Are there any other factors that could affect the results of the experiment in general? We chose a system with a single processor to eliminate an unfair environmental factor. The version of JPF used in our experiments does not include a multi-threaded state space search while testing with ConTest can take advantage of multiple processors. Therefore, conducting the experiment in a multi-processor environment would provide ConTest with an unfair advantage in terms of efficiency. In the future we would like to compare ConTest with a version of JPF that uses a parallel randomized state-space search [9].

We chose a single user machine because we will use real time instead of CPU time as the primary measure of efficiency. We use real time because ConTest utilizes random delays (e.g., sleep()) which are not captured when measuring only CPU time. In a multi-user environment we can not control the effect of other user processes on the analysis times of ConTest and JPF.

5 Experimental Procedure

Our procedure for comparing the fault detection capabilities of ConTest and JPF involved three main steps, which are repeated for each example program:

1. *Mutant generation:* A subset of the *ConMAn* mutation operators for concurrent Java are applied to an example program to generate mutants. Each mutant is the example program with one syntactic change. Table 3 provides details on the number mutant generated for our example programs.
2. *Analysis:* The analysis step is conducted for both ConTest and JPF. We will now describe the analysis with reference to examples of using ConTest. First, we analyze the example program to determine the expected observable output. The expected output could include any output generated by the program or the analysis technique. For example, with ConTest we include the standard command-line output and the standard error produced by any exceptions. After obtaining the expected output we analyze each mutant program and compare the mutant output with the expected output. An example of comparing the output of the mutant with the original program would be to use the diff program under Linux to compare the output of one execution of a mutant using ConTest with the expected output. It is possible that before comparing the output it may have to be normalized. For example, with ConTest the output was sorted to account for different interleavings of the concurrent example programs. The *Analysis* process for ConTest and JPF were conducted sequential.
3. *Merge and display of results:* We compare the analysis results of ConTest and JPF to determine which tool is more effective and efficient. To determine which technique is more effective we compare the mutant scores and the ease to kill kinds of mutants by each technique. To determine which technique is more efficient we compare the cost to kill a mutant by each technique.

ConMAn Op	Tickets-OrderSim	Linked-List	Buf-Writer	Account-Program
MXT	0	0	0	0
MSP	0	0	3	1
RTXC	0	0	2	1
RNA	0	0	0	0
RJS	0	0	1	0
ASTK	0	0	0	0
RSTK	0	0	0	0
ASK	0	0	3	1
RSK	0	0	0	3
RSB	1	2	3	2
RVK	0	0	0	0
SHCR	1	0	0	0
SKCR	0	0	3	0
EXCR	0	0	0	0
SPCR	1	2	3	1
TOTAL	**3**	**4**	**18**	**9**

Table 3. The number of mutants generated for each example program

6 Experimental Outcome

6.1. Effectiveness

There are two possible outcomes for the effectiveness of ConTest and JPF on our example programs. Both tools might be alternatives and capable of finding the same mutant faults or both tools might be complementary and be beneficial to use in combination. To assess the effectiveness

Example Program	No. of Mutants Generated	ConTest		JPF		ConTest+JPF	
		Mutants Killed	Mutant Score	Mutants Killed	Mutant Score	Mutants Killed	Mutant Score
TicketsOrderSim	3	3	100%	3	100%	3	100%
LinkedList	4	2	50%	2	50%	2	50%
BufWriter	18	7	38.9%	9	50%	9	50%
AccountProgram	9	7	78%	5	56%	7	78%
TOTAL	34	19	56%	19	56%	21	62%

Table 4. The mutant scores of ConTest, JPF and ConTest+JPF for each example program

of ConTest and JPF at detecting faults we use the results of the mutant score and ease to kill measurements.

Using the *ConMAn* operators we generated a total of 34 mutants for our four example programs. Both ConTest and JPF were able to detect 19 mutants each (a mutant score of 56%). The mutant score for each of the four example programs is given in Table 4. The reason the mutant scores are not higher is that some of the mutants may not be detectable using the test inputs and properties used in the experiment and some mutants may be equivalent. We have chosen to leave in these mutants because the identification of equivalent mutants is undecidable and estimating if a mutant is equivalent for concurrent programs is very difficult. In Figure 4 we provide a more detailed view of how mutant faults were detected by ConTest and JPF. Most of the faults in both tools were detected by assertion violations. However one interesting thing to note is that ConTest was able to detect a deadlock that was not detected by JPF. The reason there were not more deadlocks detected by both tools is that the example programs were all small in size and typically did not contain nested critical regions. Mutation of nested critical regions is most likely to produce mutants that will cause deadlock.

To determine if the distribution of mutants killed and not killed by ConTest and JPF is the same we used a chi-squared test. The null hypothesis for the test was: measurements from ConTest and JPF are from the same distribution. The test produced a low χ^2 value (0.06) indicating that at the standard confidence level of 0.05 we can not reject the null hypothesis. Although the percentage of mutant faults detected by both tools is identical we have still not determined if ConTest and JPF are alternative or complementary for our example programs.

To assess if ConTest and JPF are alternatives or complementary we need to consider the ease to kill and the ability of each tool to detect different types of mutants. Figure 5 is a bar graph which shows the percentage of mutants generated by each *ConMAn* operator that are killed by ConTest and JPF. The graph shows that there are some variations in the ability of ConTest and JPF at the mutant operator level. In particular ConTest found a higher percentage of ASK and MSP mutants while JPF found a higher percentage of RJS and RSB mutants. For all other types of mutants both tools found the same percentage. Figure 6 provides an analysis of the number of mutants detected by both tools, one tool or neither tool. Of the 34 mutants, 17 (50%) were detected by both tools, 2 (6%) were detected by ConTest

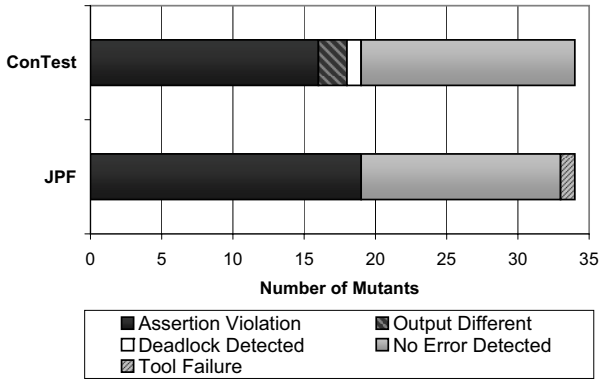

Figure 4. Detailed mutant results

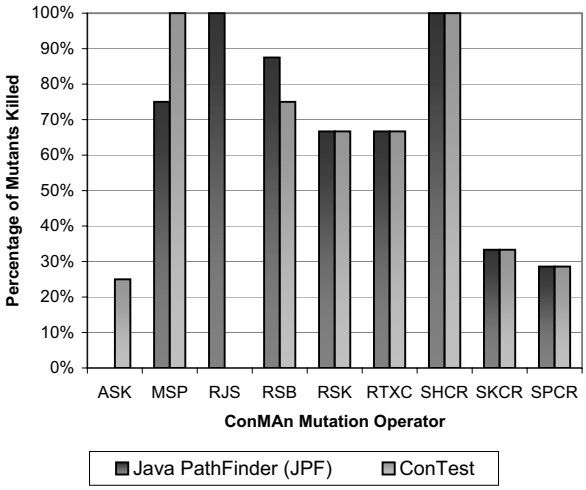

Figure 5. Ease to kill each kind of mutant

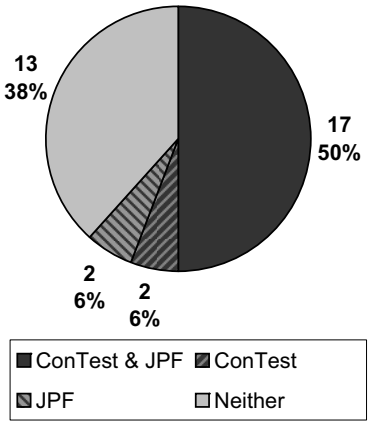

Figure 6. Mutants killed by both, one or neither tool

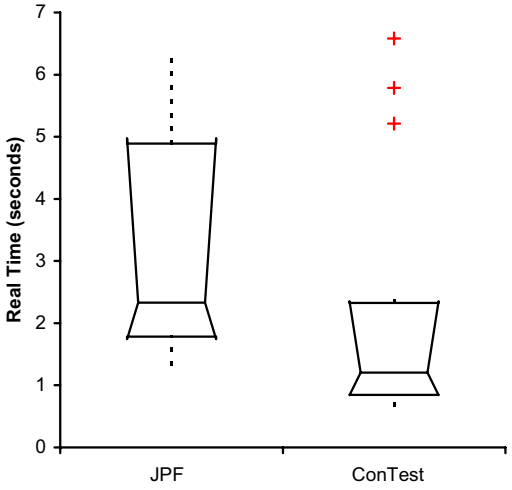

Figure 7. Box Plots of cost to kill mutants for ConTest and JPF

only, 2 (6%) were detected by JPF only and 13 (38%) were detected by neither tool. The mutant score of using ConTest and JPF in combination is 62%, 6% higher than using either approach in isolation. The improved mutant score overall seems to indicate that ConTest and JPF are complementary and their combined usage is beneficial for the example programs. However, when we consider the mutant scores for each example program the two tools seem to be alternatives. Table 4 shows the mutant scores for each program. In both the LinkedList and the TicketsOrderSim programs the mutant scores are the same for ConTest and JPF together and in isolation. For the AccountProgram, ConTest was able to detect 2 more mutants (ASK, MSP) and for the BufWriter program, JPF was able to detect 2 more mutants (RJS, RSB). Therefore, for each example program the combined use of ConTest and JPF achieved the same mutant score as the better of the two tools in isolation indicating that for our example programs ConTest and JPF are *alternatives*.

6.2. Efficiency

There are 3 possible outcomes with respect to the efficiency of ConTest and JPF at detecting faults in our 4 example programs. ConTest or JPF might be more efficient, there may be no difference in the efficiency or there may be a mixed result. To determine the efficiency outcome we used the cost to kill a mutant. That is, for all mutant faults detected by both ConTest and JPF we compared the real time to detect a mutant for each tool.

To visually compare the cost to kill a mutant we present side-by-side box plots in Figure 7. For each box plot the middle line in the box indicates the median value and the ends of the box are the first and third quartile. From the box plots it appears that ConTest is more efficient at detecting a fault when both techniques are capable of finding the fault. Specifically, ConTest has a smaller median and smaller first and third quartiles.

Based on the box-plot we tested the proposition that the cost to kill a mutant using JPF was less than using ConTest. We tested the proposition using a 1-tailed paired t-test[3]. We were able to conclude that at the 0.05 level our proposition was not correct (p-value = 0.0085).

In summary the results of our box plot and paired t-test conclude that ConTest is *more efficient* than JPF for our example programs.

7 Threats to Validity

There are a number of issues of validity to be considered: internal validity, external validity, construct validity, and conclusion validity[23].

Internal validity. Threats to internal validity are *"...influences that can affect the independent variable with respect to causality, without the researcher's knowledge"* [23]. There is a clear history regarding the causal relationship between fault detection tools (the independent variable) and the number of faults detected (the dependent variable). This history limits the possibility of threats to

[3]In order to perform a paired t-test an important assumption is that the difference between the test data must by normally distributed. That is, the difference in detection times for each mutant using ConTest and JPF must be normal. We used the Shapiro-Wilk test to assess normality.

internal validity.

External validity. Threats to external validity are *"...conditions that limit our ability to generalize the results of our experiment..."* [23]. In our experiment there are three major threats to external validity. First, a threat to external validity is possible if the mutant faults used do not adequately represent real faults for the programs under experiment. We ensure representative mutants by using the *ConMAn* mutation operators which are based on an existing fault model [14]. Second, a threat to validity is possible if the software being experimented on is not representative of the software to which we want to generalize. In our experiment the small set of programs are not representative of all concurrent Java applications and therefore our results do not generalize well. Third, an additional threat to the validity is that the configurations of JPF and ConTest used in our experiment limit our ability to generalize to each approach. For example, JPF can be customized with other search algorithms and scheduling strategies that may affect both its effectiveness and efficiency with respect to fault detection. In a recent study, Dwyer, Person, and Elbaum concluded that the search order used in a tool can influence the effectiveness of the analysis [10].

Construct validity. Threats to construct validity are *"...concerned with the relation between theory and practice"* [23] and *"...refer to the extent to which the experimental setting actually reflects the construct under study"* [23]. There is a potential for threats to construct validity if ConTest and JPF are not used in the way in which they are intended. We discussed this issue briefly when we outlined the importance of selecting tools with similar goals that are applied to the same kind of applications. Ensuring this is the case limits the need to modify how the tools are used.

Conclusion validity. Threats to conclusion validity are *"...concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment"* [23]. In order to ensure conclusion validity in our experiment we need to have confidence that our measurements are correct and the statistical tests are used correctly. First, we ensure that our measurements are recorded correctly by automating the collection of measurements using our *ExMAn* framework. Second, in order to ensure that the statistical tests used to evaluate the measurements allow for correct conclusions we have done our best to ensure that none of the statistical test assumptions are violated.

8. Related Work

Several empirical studies have focused on the ability of testing and model checking to find faults. For example, a case study involving testing and model checking was conducted by Chockler et al., who compared ConTest and the ExpliSAT model checker using two real programs at IBM [6]. The results of the case study focused on the usage and the comprehensiveness of the results of each tool. Overall, ConTest was found to be easier to use but was not as a comprehensive in identify potential problems in the software. The comprehensiveness considered by Chockler et al. is a similar measurement to our effectiveness. One difference between this research and our own is that Chockler et al. do not focus on the efficiency of each tool.

Brat el al. conducted a controlled experiment involving traditional testing, runtime analysis, model checking and static analysis [5]. The experiment involved human participants using the different techniques to detect 12 seeded faults in NASA's Martian Rover software. The kinds of faults included deadlock, data races and other non-concurrency faults. Although no statistically significant conclusions were drawn from the experiment the authors stated that the results *"...confirmed our belief that advanced tools can out-perform testing when trying to locate concurrency errors"* [5]. We achieved a different outcome in our experiment that compared the same model checker (JPF) with testing. We believe the primary difference for our results is the kind of testing used. On the one hand, we used testing with ConTest, which is a sophisticated tool that automatically seeds delays into Java byte code to explore different interleavings. On the other hand, Brat el al. used standard black box system testing. Exploration of different interleavings was not automatic – instead the testing relied on native scheduling differences by using different operating systems with different Java Virtual Machines. Furthermore, manual instrumentation of delays as well as thread priorities were used.

Our work differs from this previous work in that we are able to draw statistically significant conclusions regarding both the effectiveness and efficiency of testing (with ConTest) and model checking (with JPF) at finding mutant faults. Although the previous comparisons did not have statistically significant quantitative results it is important to acknowledge their contributions to the community and to our own work since we build upon these previous studies.

9. Conclusion

We have presented a controlled experiment that uses program mutation to compare the fault detection capabilities of testing with ConTest and model checking with JPF. The experimental procedure is automated using our *ExMAn* framework and the use of mutation with concurrent Java is supported by our *ConMAn* operators. The experiment demonstrates the feasibility of program mutation as an aid to understanding testing and model checking of concurrent software.

Our experiment has tried to better understand the effec-

tiveness and efficiency of ConTest and JPF at detecting mutant faults in our example programs. With respect to effectiveness, we conclude that ConTest and JPF are most likely *alternative* fault detection techniques rather than complementary. However, the ease to kill measurements show that both tools do not detect all of the same kinds of mutants equally. With other example programs there maybe a potential to use ConTest and JPF in a complementary way. With respect to efficiency, we conclude that ConTest is more efficient and can kill a mutant in less time on average than JPF for our example programs.

It is important to be clear that the results of our experiments do not generalize to all concurrent Java software. The example programs are relatively small in size and do not contain any of the new J2SE 5.0 concurrency mechanisms. However, the results still provide insight into the relationship between testing with ConTest and model checking with JPF and the usefulness of each in detecting bugs and improving the quality of concurrent software. In order to achieve stronger and more general conclusions we need to conduct further experiments. We need to compare testing with ConTest and model checking with JPF using larger sets of example programs and differemt configurations of each tool in order to build a strong body of empirical results.

References

[1] Java PathFinder website. Web page: `http://javapathfinder.sourceforge.net/`.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. of* 27^{th} *International Conference on Software Engineering (ICSE 2005)*, pages 402–411, May 2005.

[3] J. S. Bradbury, J. R. Cordy, and J. Dingel. ExMAn: A generic and customizable framework for experimental mutation analysis. In *Proc. of the* 2^{nd} *Workshop on Mutation Analysis (Mutation 2006)*, pages 57–62, Nov. 2006.

[4] J. S. Bradbury, J. R. Cordy, and J. Dingel. Mutation operators for concurrent Java (J2SE 5.0). In *Proc. of the* 2^{nd} *Workshop on Mutation Analysis (Mutation 2006)*, pages 83–92, Nov. 2006.

[5] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Păsăreanu, A. Venet, W. Visser, and R. Washington. Experimental evaluation of verification and validation tools on Martian Rover software. *Formal Methods in Systems Design Journal*, 25(2-3):167–198, Sept. 2004.

[6] H. Chockler, E. Farchi, Z. Glazberg, B. Godlin, Y. Nir-Buchbinder, and I. Rabinovitz. Formal verification of concurrent software: Two case studies. In *Proc. of the Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD-IV)*, pages 11–21, Jul. 2006.

[7] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proc. of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT '98)*, pages 48–59. ACM Press, 1998.

[8] H. Do and G. Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *Proc. of the* 21^{st} *IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 411–420, 2005.

[9] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *Proc. of the* 29^{th} *International Conference on Software Engineering (ICSE 2007)*, pages 3–12, May 2007.

[10] M. B. Dwyer, S. Person, and S. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *Proc. of the* 14^{th} *ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*, pages 92–104. ACM Press, Nov. 2006.

[11] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[12] Y. Eytani, E. Farchi, and Y. Ben-Asher. Heuristics for finding concurrent bugs. In *Proc. of the* 1^{st} *International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2003)*, Apr. 2003.

[13] Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In *Proc. of the* 2^{nd} *International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2004)*, Apr. 2004.

[14] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proc. of the* 1^{st} *International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2003)*, Apr. 2003.

[15] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), Apr. 2000. Special issue containing selected submissions for the 4^{th} SPIN Workshop.

[16] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[17] B. Long, D. Hoffman, and P. Strooper. Tool support for testing concurrent Java components. *IEEE Trans. on Soft. Eng.*, 29(6):555–566, Jun. 2003.

[18] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, 1992.

[19] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proc. of the* 9^{th} *European Software Engineering Conference held jointly with the* 11^{th} *ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-11)*, pages 267–276. ACM Press, 2003.

[20] J. Rushby. Disappearing formal methods. In *Proc. of the High-Assurance Systems Eng. Symp. (HASE'00)*, pages 95–96, Nov. 2000.

[21] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[22] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, Apr. 2003.

[23] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Software Engineering. Kluwer Academic Publishers, 2000.