Compiling a Benchmark of Documented Multi-threaded Bugs

Yaniv Eytani
Computer Science Department,
Haifa University
ieytani@cs.haifa.ac.il

Shmuel Ur
IBM Research Lab in Haifa,
Haifa University Campus
ur@il.ibm.com

Abstract

Testing multi-threaded, concurrent, or distributed programs is acknowledged to be a very difficult task. We decided to create a benchmark of programs containing documented multi-threaded bugs that can be used in the development of testing tool for the domain. In order to augment the benchmark with a sizable number of programs, we assigned students in a software testing class to write buggy multi-threaded Java programs and document the bugs.

This paper documents this experiment. We explain the task that was given to the students, go over the bugs that they put into the programs both intentionally and unintentionally, and show our findings. We believe this part of the benchmark shows typical programming practices, including bugs, of novice programmers.

In grading the assignments, we used our technologies to look for undocumented bugs. In addition to finding many undocumented bugs, which was not surprising given that writing correct multi-threaded code is difficult, we also found a number of bugs in our tools. We think this is a good indication of the expected utility of the benchmark for multi-threaded testing tool creators.

1. Introduction

The increasing popularity of concurrent programming on the Internet as well as on the server side has brought the issue of concurrent defect analysis to the forefront. Concurrent defects such as unintentional race conditions or deadlocks are difficult and expensive to uncover and analyze, and such faults often escape to the field.

One reason for this difficulty is that the set of possible interleavings is huge, and it is not practical to try all of them. Only a few of the interleavings actually produce concurrent faults. Thus, the probability of producing a concurrent fault is very low. Another problem is that since the scheduler is deterministic, executing the same tests many times will not help, because the same interleaving is usually created. This is true for simple tests, regardless of the environment, and for tests of average complexity re-executed in a similar environment. The problem of testing multi-threaded programs is compounded by the fact that tests that reveal a concurrent fault in the field or in stress test are usually long and run under different environmental conditions. As a result, such tests are not necessarily repeatable, and when a fault is detected, much effort must be invested to recreate the conditions under which it occurred.

The first Parallel and Distributed Systems: Testing and Debugging (PADTAD) workshop, as well as [1], discussed the need for creating a benchmark that contains multi-threaded Java programs with documented bugs related to synchronization. This benchmark will be useful for the evaluation and improvements of testing tools in the parallel and distributed testing domain. We need a benchmark that is comprehensive enough to contain all the common multi-threading bug patterns [2][3][4][5], as we expect no silver bullet and probably different approaches will be better for different bug patterns.

Toward this goal, students of an undergraduate software testing class were given an assignment to write programs containing one (or more) concurrent bugs. Clearly, that the programs created by the students are biased toward bugs typical to novice programmers. However, this work is not meant as the full benchmark but as a part of it. Larger programs made by more experienced programmers will be taken mainly from open source applications. We hope to use the bug reports to identify problems in old versions and add the versions with the documented bugs into the benchmark.

As part of the course, the students studied the taxonomy of concurrent bug patterns [2] and technologies for finding such bugs [6][7][8][9][10][11]. They were told the assignment was to write a program (or find and modify an existing program) that has at least one multi-threaded bug. Because the course stressed documenting bugs as an important part of the tester work [12], this exercise was a relevant practical experience. All the bugs in the program were to be documented and points would be taken off for every undocumented bug found. The students were given a standard wrapper format used to

Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

run the program. They were also asked to write a report describing the program and its functions, bugs, and possible outcomes.

The students were offered extra credit for testing their program with raceFinder [13] (using ConTest[7] instrumentation), a tool that uses heuristics to increase the probability of manifesting concurrent bugs, and writing a report about the test.

The course included presentations about bug patterns in multi threading [2] and bug taxonomies [14]. A common bug pattern, called the lost notify, is when the notify is done before the wait and therefore is not used. When the wait is executed it may wait forever. At first glance, it may seem that there is a problem with the methodology of asking the students to use these bug patterns, as only bugs belonging to the patterns taught will be part of the benchmark. However we are convinced this is not a problem as the bug pattern work is quite comprehensive. Indeed, this risk exists in all bug taxonomies and inspection checklists - if something is not included, it may not be seen. The rationale that makes this risk acceptable is that this method makes the work more efficient. Common enough bugs will eventually make it into the list.

The lessons learned from this exercise can be divided into two categories. Technically, we found some bugs in our tools, mainly because the students programmed in ways we never even considered. We also found bugs that our tools were unable to discover, and we learned about the bugs' novice programmers create. Methodologically, we can now define the students' assignment more clearly, and we also have a better idea how to further expand the benchmark.

2. Interaction with students

The assignment to write a program with documented bugs was given to students in a software testing course. The assignment, which appears on the class Web page (http://cs.haifa.ac.il/courses/softtest/testing2003/), was defined as follows:

- Create a program with documented bugs.
  - The program should perform some potentially useful task (i.e. there should be some rational for the code)
  - The program should include intermittent (appearing irregularly) bugs that are caused by multi-threading.
  - The program should have two parameters as inputs:
    - The name of the output file.
    - Parameter of concurrency {little, average, lot} (optional). This parameter was program-dependent. The idea was that if the bug is dependent on the number of threads, or on the number of races, little will create small number of threads and few races such that the likelihood of bugs will be low; while lot will increase the likelihood of bugs to such a degree that they will most likely appear.
  - Have the program output tuples of the form into the output file <program name, result, bug pattern indicated>. (If terminated successfully, the bug pattern should be none)
  - Create a README file explaining the program and its function; the bug and its location (lines of code); the bug pattern to which the bug conforms; and which variables are involved and should be attached.
  - Create a dictionary file containing all the possible tuples <program name, result, bug pattern indicated>. It is possible that one result will indicate more than one bug pattern.

The students did not totally understand the goal of the assignment (compilation of a benchmark), the nature of the program they were supposed to write, and the correct input and output interface. Our guess they did not fully realized the homework assignment was actually going to be used, and that therefore the requirements are not arbitrary. Some of the students did not implement the right output interface even after a second and third explanation. Most of the students didn't submit clear running instructions for the program. The running environment was not defined in the assignment requirement, and some submitted projects that could run only on a proprietary environment (e.g., the account program that must be run through Borland Jbuilder).

The students who used raceFinder were asked to write a research report. They were also asked to try the different features of raceFinder, evaluate its effectiveness in manifesting a concurrent bug of the pattern in the student's program, and explain their findings. The report included background on the specific bug pattern, the interleaving that manifested the bug, and statistics comparing raceFinder features to the bug manifesting. Sometimes the results were compared with manifesting the bug manually by inserting yield() and wait() primitives and re-compiling the program.

3. Bugs created for the benchmark

Most of the students chose to write non-atomic bug patterns, as can be seen in figure 1, mainly missing a synchronized statement that leads to unforeseen interleaving by the programmer (including data races). Other bug patterns used include deadlock, the sleep bug pattern and the orphaned thread bug pattern. All of those bug patterns were given as a presentation to the students in class [2]. Few of the students created bugs that were dependent on interleavings that happen only with specific inputs.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

The students wrote mostly short or very short programs (100 – 600 lines of code), and most of that code was related to the bug. The programs generally preformed some task designed to manifest the concurrent bug in some situation. The portion of the code not related to the bug was not very large.

The input for most of the programs was created randomly, so the students did not pay too much attention to the results section of the output. They reported either a meaningless result, a binary result (a right and wrong result), or a variation, in which the output was divided into a class of right and a class of wrong results.

For example, below is the output tuples from the account program (this program simulates money transfer between different bank accounts):

1. <Account program, All amounts are 300, None>
2. <Account program, There is an amount with more than 300 and there is an amount with less than 300, No Lock>
3. <Account program, There is an amount with more than 300, No Lock>
4. <Account program, There is an amount with less than 300, No Lock >

In this example, there are four different output types. The output is a variation of a binary output and has no meaning for the user, other than to indicate whether or not the program executed correctly. A good output would have been useful in measuring the effectiveness of the testing tool in uncovering the concurrent bug, for example, how often the bug manifested.

3.1. Unintentional bugs inserted by the students

Several unforeseen interleavings were the result of the students' poor understanding of concurrent programming, and their clear understanding of only the buggy interleaving they intended the program to manifest. For example the programs: shop, buggy program, and allocation vector had many interleavings that were not documented in the report, which led to the same buggy result that the students intended to manifest. Some of the students did not grasp those buggy interleavings, even after we explained the problem.

Many of the students did not write the program from scratch, but chose a program that they, or someone else, had written before. Some of the students searched the Internet for buggy programs to use for the exercise. However, all of them had to write a wrapping that would make the program behave according to the specification. As a result, the wrapping was always original code written by the students. Most of the students made mistakes in the wrapping code. Most did not check that the inputs conformed to specification. Many did not write the outputs correctly. Some left unused code in parts of the program. The exercise was an important part of the grade. Even though the students knew that the exercise would be graded according to the amount of bugs, and even though they did not have to fix the bugs but just document them, almost none of them created a clean program. None of the students decided, when working on the assignment and finding unintentional bugs, to document them rather then fix them. We hypothesized two possible reasons. Lucking experience the students may have thought that fixing is easier. Alternatively they may not have wanted others to see their mistakes.

4. Noisemaking, insight gained

Using raceFinder to produce test reports, and working with a large number of different users allowed us to study this tool more, better understand how to use it, and discover its current capabilities and limitations. Using raceFinder with a large body of programs containing bugs proved to be a valuable experience as we had to reason about different bugs and the interleavings that caused them, since most of the students did a poor job recording all of the interleavings that caused a bug to appear. We learned a few important lessons from this:

- Given a proper explanation, students can fine-tune raceFinder options to find their concurrent bugs. ConTest was built with the notion that it should be transparent to the user. raceFinder is built with the notion that automated testing and debugging is not yet feasible, and the user should assist this process. Future work on raceFinder will try to combine both approaches to achieve better usability, while continuing to achieve effective results

- Comparing manual noise creation at the right spot with raceFinder heuristics shows that there is still room for improvement in the heuristics precision. This fact, based on the raceFinder reports, support the hunch that creating noise in a few places is more effective for finding concurrent bugs, than creating noise in many program locations. For example, in some of the programs, adding a Yield() statement in the right program location causes the bug to manifest at a high rate. In contrast, raceFinder can achieve this manifestation rate when applying a high noise level that causes a number of Yield() statements to be inserted in a few program locations. It is advisable to apply more intelligent heuristics that could use static analyses [15][16], dynamic analyses [17][18], or both [19] to reduce the number of program locations. This idea is now being implemented in raceFinder with good results.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

Figure 1. The benchmark programs

	Program name	Functionality	code lines	Output	Bug name	bug type	Use RF
1	SoftWareVerification	Bubble-sort	362	Correct	Not-atomic, Orphaned-Thread	not-atomic, Orphaned-Thread	no
2	Test	Allocation vector	286	Correct	Weak-reality (two stage access)	not-atomic	yes
3	SoftTestProject	Selling airline tickets via agents	95	Wrong	Interleaving	not-atomic	no
4	BuggedProgram	thread pingPong	272	Wrong		not-atomic	no
5	ExcBug	Producer–consumer	209	Correct	Weak reality [wrong \ No-Lock]	not-atomic	no
6	bufwriter	Buffer read-write	255	Correct-	[Wrong lock/No lock]	not-atomic	no
7	BugGen	Client–Server relationship	161	Correct	Liveness – Bug (Dormancy)	?	no
8	bug1	Copying files	121	Correct	Deadlock	deadlock	yes
9	BuggyProgram	Generates random numbers	359	Correct	Not-atomic, Wrong-Lock or No-Lock, Blocking-Critical-Section	Not-atomic	yes
10	GarageManager	Workers tasks	584	Correct	Blocking-critical-section	deadlock	no
11	Loader	Bubble sort	130	Wrong	Initialization-Sleep Pattern	sleep	no
12	Account	Bank	155	Correct	[Wrong Lock/No Lock]	not-atomic	yes
13	IBM_Airlines	Producer–consumer	243	Correct	[Condition-For-Wait]	not-atomic	yes
14	ThreadTest	Processors performance benchmark	249	Correct	Blocking_critical_section	deadlock	yes
15	BugTester	Linked list	416	Wrong	Non-atomic	not-atomic	no
16	ProducerConsumer	Producer–consumer	203	Wrong	Orphaned-Thread	Orphaned-Thread	no
17	TicketsOrderSim	Selling airline tickets via agents	183	Correct	Double checked locking	DCL	yes
18	shop	Simulates the operation of shop	273	Correct	Sleep , weak reality (lock unlock lock)	not-atomic	yes
19	BoundedBuffer	Producer–consumer	536	Correct	Notify instead of notifyAll	notify instead of notifyAll	yes
20	MergeSortBug	Mergesort	257	Correct	Not-atomic	not-atomic	no
21	MergeSort	Merge sort	375	Wrong		not-atomic	no
22	Manager		236	Wrong	Not-atomic	not-atomic	no
23	Maximum	Calculating maximum distributively	167	Correct	Guarded (if statement) data race	not-atomic	yes

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

- Testing more programs with non-atomic bug patterns supports the claim that raceFinder can effectively handle this type of bug pattern, and that raceFinder's current heuristics can uncover non-atomic buggy interleavings.

- Writing multi-thread code is hard—even the small programs contained undocumented bugs and buggy interleaving. It was sometimes hard even for us, despite our previous experience with concurrent bugs, to understand the undocumented buggy interleavings (even with small programs).

4.1. Lessons learned from the bug patterns

A large number of programs containing concurrent bugs, some of which were conceived in surprising ways, allowed us to reason about the factors that contribute most to the design and implementation of programs that contain concurrent bugs. There are three main factors upon which a manifestation of a buggy interleaving depends:

- The scheduling policy of the JVM – this policy is usually deterministic and thus produces a limited subset of interleaving space [20][7].

- The input of the program (control flow) – some of the interleavings that induce a concurrent bug are input-dependent. It is not enough to use a noise- dependent programs the students created for the benchmark was random; hence, raceFinder had little effect in raising the bug manifestation rate.

- The design of the program contains a fault (not the implementation) and this fault manifests only in rare circumstances requiring complex scenarios. In such cases while, noise could theoretically cause the buggy interleaving to appear, it may remain very rare.

Another less important factor is the two levels of the Java memory model [22].

We saw a number of bugs that cannot be uncovered with tools such as ConTest or raceFinder. Furthermore, some of these bugs may actually be masked when you look for them with these tools. Both tools instrument at the bytecode level, which imposes inherent limitations. For example, if a bytecode is not atomic, the tools cannot create noise inside that bytecode. Bugs that have to cope with scheduling inside the JVM—for example, the JVM definition of the order in which the waiting threads are awakened by a notify—cannot be impacted. In addition, some bugs, notably bugs related to two-tier memory hierarchy, will not be observed on a one-tier memory Java implementation, regardless of the scheduling.

Intelligent noisemakers can effectively change JVM scheduling to manifest concurrent bugs. However, they cannot control the input. Additionally, good testing metrics are required when the bugs are dependent not only on a specific interleaving (or a few interleavings), but also related to specific input or inputs (if the input is random, a partial replay procedure will not help). There are tools designed to give coverage of the program's logic (for example ConAn [23]), and it may be interesting to combine a noisemaker with such tools.

5. Bugs and problems we found in our tools

In addition to the bugs in the student applications, we found bugs in our tools. Most of the bugs found were due to the non-standard programming practice used by the students that were not considered in the tool design. This section describes these bugs

5.1. Problems discovered in raceFinder

- Some bugs in the implementation (mainly the input to noise interface) of raceFinder were discovered and fixed.

- When there are many memory accesses in the program, delays (e.g., wait) caused overhead, which made it impossible to test the program. This led us to the conclusion that a specific bug pattern optimization that reduces the number of noised memory location (assuming that specific pattern causes the bug to manifests) need be added to raceFinder to allow testing of such programs.

- When running the program instrumented with raceFinder, even slight changes in the program timing can cause the program to trash and run endlessly. It is not clear yet what exactly causes this problem. (For example, the DCL program with high concurrency level runs without ending when used with raceFinder.)

- RaceFinder currently doesn't offer sufficient debugging information to give a good understanding of which interleaving caused a bug to manifest and why. Isolating the places where related noise occurred is not always enough to debug the program.

5.2. Problem that was discovered in ConTest

An interesting bug was found using ConTest. ConTest has a heuristic called halt-one-thread. When this heuristic

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

is activated, a thread is put into a loop, conditioned on a flag called progress. In the loop, progress is set to false and a long sleep is executed. In every other thread, if something happens, progress is set to true. This heuristic allows us to stay for a very long time at this point. Only when the rest of the program cannot proceed and waits for this thread will we get out of this loop. This is useful for creating unlikely timing scenarios. We reasoned that this couldn't create a deadlock—as soon as there is no progress, we would get out of the loop

We did not take into account the possibility that students (and by extension, other programmers) might implement a wait using a busy loop. When a tread waits on a condition, instead of doing a wait, it executes a loop in which it continuously asks if the condition happened. In such a case, the thread that entered the halt-one-thread will never extract itself, as there is "progress" outside. The other thread will forever wait (busily), and nothing will progress.

6. Adding more students programs to the benchmark

Our benchmark is off to a good start with almost 40 programs, about half of which where created by students. We would like to extend the benchmark with additional student programs. Hopefully, other universities will join the effort. This section details our finding and direction with respect to running such a homework assignment in the future.

6.1. Future work on expanding the benchmark

- The students should write larger programs with code that perform tasks not related directly to the concurrent bug to allow testing the scalability of the testing tools. We may also ask the students to insert a concurrent bug into a large existing multi-thread program (which will probably lead to new unforeseen concurrent bugs in that program).

- The benchmark should be balanced, having at least a few programs from each bug pattern. This will allow us to derive more conclusive results about each bug pattern.

- The students should get longer lectures regarding multi-threading issues, the motivation behind compiling the benchmark, and what they are expected to do.

- Because one of the benchmark's goals is to encourage the integration of several testing techniques and increase the testing success rate, the students should use more than one testing tool (and tool type), and explore the integration of these techniques.

- Giving the students a standard running environment for testing their programs should be considered. By allowing the students to work on running environments that are not generally available to them, we can improve the testing process, and dedicate more time to testing the programs. This would also allow us to compare the results from the different programs tested on the same testing setup.

6.2. Defining the benchmark standards

From our experience with creating a benchmark of a multi threaded programs, we obtained a clearer understanding on how to define the benchmark standards. In the benchmark, the following topics should be defined*:*

- The input and the output interface. In addition to documentation, we should provide APIs in order to reduce customization work.

- The structure of the submitted report. The report should including the program execution orders and comprehensive description about its structure. An example and a template should be made available.

- Creation of a standard for a trace, and addition of a trace to the programs in the benchmark to allow tools that use this resource to work on the programs in the benchmark.

- The program should include a standard option that allows the program to receive input from a file; this option should allow using the program with a testing metric and a replay procedure to automatically correlate inputs with bugs and coverage.

7. Conclusion

Our undergraduate software testing class students were given an assignment to write a potentially useful program, i.e. one that does something, containing one (or more) concurrent bugs. They were told that additional bugs are permissible, but that all the bugs in the program should be documented. They were also asked to write a report describing the program and its functions, bugs, and possible outcomes. The assignment turned out to be more difficult then expected. The open nature of the assignment "create a useful program with an interesting bug" confused quite a number of students. The requirement that all bugs be documented - which meant that instead of correcting the program after it sort of worked, they could have just documented the additional

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

bugs - was used by none. All the students submitted programs in which the only documented bugs where the bugs that they put in by design. It is a natural reaction for a programmer to try to hide his mistakes as much as possible - showing your bugs goes against the grain. However, it has been shown many times and in many domains [24] that the first step in improving is taking your mistakes seriously and analyzing them. Our first conclusion is that even though we stressed this point for the entire course we did not drive it home.

In testing the homework assignments we found some bugs in our tools, mainly because the students programmed in ways we never even considered. Testing tool creation follows a pattern: you see a bug, figure an automatic way to detect it, and create or augment a tool to do it. For a tool maker, it is beneficial to be exposed to different programming practices, different styles, and to many programming guidelines. The assignments represent quite a large number of bugs, written in a variety of styles, and therefore useful for the purpose of evaluating testing tools. There is a bias toward the kind of bugs that novice programmers create. A good source for bugs created by experience programmers is the open source code. One way to collect such bugs for the benchmark is to follow the bug fix annotation and ask the owners for the source code containing the bug. This way you will have the bug and the correct fix for the benchmark which will be useful to check on some testing tools comments.

We saw a number of bugs that cannot be uncovered with tools such as ConTest or raceFinder. Furthermore, some of these bugs may actually be masked when you look for them with these tools. Both tools instrument at the bytecode level, which imposes inherent limitations. For example, if a bytecode is not atomic, the tools cannot create noise inside that bytecode. Bugs that have to cope with scheduling inside the JVM, for example, the JVM definition of the order in which the waiting threads are awakened by a notify, cannot be impacted. In addition, some bugs, notably bugs related to two-tier memory hierarchy, will not be observed on a one-tier memory Java implementation, regardless of the scheduling. The benchmark contains many bugs and we are certain that no one tool can find all of them. By trying to uncover the bugs with the different tools, we will enhance the tools to detect more bug types, and figure out the correct mix of tools to use for efficient verification

In the process, we have learned about creating benchmarks in general, and creating benchmarks using student assignments in particular. We now know how to define the students' assignment more clearly, and where the pitfalls are expected to be. As a result of our preliminary use of the benchmark, we also have a better idea how to further expand the benchmark. This is an ongoing work in which the benchmark is expanded and more features are added. In the near future, we expect to get additional feedback from benchmark users which we will use in the next iterations of this exercise.

8. Acknowledgments

We would like to thank Eitan Farchi for his guidance in the development of raceFinder, and the Caesarea Edmond Benjamin the Rothschild Foundation Institute for Interdisciplinary Applications of Computer Science, at the University of Haifa for partial funding of research related to raceFinder.

9. References

[1] Klaus Havelund, Scott D. Stoller, and Shmuel Ur. "Benchmark and framework for encouraging research on multi-threaded testing tools." In Proceedings of the Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD), April 2003.

[2] Eitan Farchi, Yarden Nir, and Shmuel Ur. "Concurrent Bug Patterns and How to Test Them." In International Parallel and Distributed Processing Symposium, IPDPS 2003, PADTAD Workshop, 2003.

[3] The Java Developer Connection Technical Tips, dated March 28, 2000, at http://developer.java.sun.com/developer /TechTips/2000/tt0328.html.

[4] Lewis and D. J. Berg. Threads Primer. Prentice Hall, Englewood Cliffs, NJ (1996).

[5] D. Lea. Concurrent Programming in Java, Second Edition. Addison-Wesley Publishing Co., Reading, MA (2000).

[6] S. Savage. “Eraser: A Dynamic Race Detector for Multithreaded Programs.” ACM Transactions on Computer Systems 15, No. 4, 391–411 (1997).

[7] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. "Mulithreaded Java Program Test Generation." IBM System Journal, Vol. 41, No. 1, 2002.

[8] Scott D. Stoller. "Testing Concurrent Java Programs Using Randomized Scheduling." In Proceedings of the Second Workshop on Runtime Verification (RV), volume 70(4) of Electronic Notes in Theoretical Computer Science. Elsevier, 2002.

[9] K. Havelund and T. Pressburger. "Model checking Java programs using Java pathfinder." International Journal on Software Tools for Technology Transfer, STTT, 2(4), April 2000.

[10] S. D. Stoller. "Model-checking multi-threaded distributed Java programs." International Journal on Software Tools for Technology Transfer, 4(1):71–91, Oct. 2002.

[11] Eitan Farchi, Alan Hartman, and Shlomit S. Pinter. "Using a model-based test generator to test for standard conformance." IBM System Journal, Vol. 41, No. 1, 2002.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

[12] Kaner, Cem. "Bug Advocacy" (July 2000) See http://http://www.testingcraft.com/bug-advocacy-kaner.pdf

[13] Y. Ben-Asher, Y. Eytani, and E. Farchi. "Heuristics for finding concurrent bugs." In International Parallel and Distributed Processing Symposium, IPDPS 2003, PADTAD Workshop, 2003.

[14] Vijayaraghavan, G. & Kaner, C. "Bug Taxonomies: Use Them to Generate Better Tests." In Software Testing Analysis & Review Conference (STAR) East 2003, Orlando, FL.

[15] Eran Yahav, "Verifying safety properties of concurrent Java programs using 3-valued logic." ACM SIGPLAN Notices, v.36 n.3, p.27-40, March 2001

[16] J.-D. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. "Escape analysis for Java." In Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Oct. 1999.

[17] Cormac Flanagan and Stephen N. Freund. "Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs." 31st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), Jan 2004

[18] Liqiang Wang and Scott D. Stoller. "Run-Time Analysis for Atomicity." In Proceedings of the Third Workshop on Runtime Verification (RV), volume 89(2) of Electronic Notes in Theoretical Computer Science. © Elsevier, 2003.

[19] J. D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. "Efficient and precise datarace detection for multithreaded object-oriented programs." In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 258–269, 2002.

[20] J. D. Choi and H. Srinivasan. "Deterministic Replay of Java Multithreaded Application." Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, ACM, New York (1998).

[21] A. Zeller. "Isolating Cause-Effect Chains from Computer Programs." Proc. ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE-10), Charleston, South Carolina, November 2002.

[22] The "Double-Checked Locking is Broken" Declaration, dated December 15, 2003, at http://www.cs.umd.edu/~pugh /java/memoryModel/DoubleCheckedLocking.html.

[23] Long, B. Hoffman, D., and Strooper, P. "Tool support for testing concurrent Java components." IEEE Transactions on Software Engineering, 29 (6), 555-566, Jun. 2003.

[24] Humphrey, Watts S. "Using a Defined and Measured Personal Software Process." IEEE Software, 77-88, May 1996.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE