Towards a framework and a benchmark for testing tools for multi-threaded programs

Yaniv Eytani[1], Klaus Havelund[2], Scott D. Stoller[3] and Shmuel Ur[4,*,†]

[1]*Computer Science Department, University of Haifa, Haifa, Israel*
[2]*Kestrel Technology, NASA Ames Research Center, Moffett Field, CA 94035-1000, U.S.A.*
[3]*Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794, U.S.A.*
[4]*IBM Haifa Research Laboratory, Haifa University Campus, Haifa, Israel*
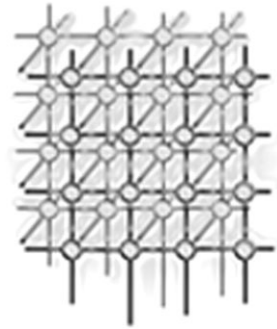
SUMMARY

Multi-threaded code is becoming very common, both on the server side, and very recently for personal computers as well. Consequently, looking for intermittent bugs is a problem that is receiving more and more attention. As there is no silver bullet, research focuses on a variety of partial solutions. We outline a road map for combining the research within the different disciplines of testing multi-threaded programs and for evaluating the quality of this research. We have three main goals. First, to create a benchmark that can be used to evaluate different solutions. Second, to create a framework with open application programming interfaces that enables the combination of techniques in the multi-threading domain. Third, to create a focus for the research in this area around which a community of people who try to solve similar problems with different techniques can congregate. We have started creating such a benchmark and describe the lessons learned in the process. The framework will enable technology developers, for example, developers of race detection algorithms, to concentrate on their components and use other ready made components (e.g. an instrumentor) to create a testing solution. Copyright © 2006 John Wiley & Sons, Ltd.

1. INTRODUCTION

The increasing popularity of concurrent programming—for the Internet as well as on the server side—has brought the issue of concurrent defect analysis to the forefront. Concurrent defects such as

---

*Correspondence to: Shmuel Ur, IBM Haifa Research Laboratory, Haifa University Campus, Haifa 31905, Israel.
†E-mail: ur@il.ibm.com

unintentional race conditions or deadlocks are difficult and expensive to uncover and analyze, and such faults often escape to the field. The development of technology and tools for identifying concurrent defects is now considered by some experts in the domain as the most important issue that needs to be addressed in software testing [1]. Having new dual core or hyper-threaded processors in the personal computer, as all processors will have, starting in 2006, makes the testing of multi-threaded programs even more important. The programs that used to work well on single-threaded and single CPU core processors are now exhibiting problems. As a result, Intel has come out with a race detection tool and Microsoft has also addressed this issue.

There are a number of distinguishing features between concurrent defect analysis and sequential testing. These differences make it especially challenging if the set of possible interleavings is huge and it is not practical to try all of them. First, only a few of the interleavings actually produce concurrent faults; thus, the probability of producing a concurrency fault can be very low. Second, under the simple conditions of unit testing, the scheduler is deterministic; therefore, executing the same tests repeatedly does not help. As a result, concurrent bugs are often not found early in the process, but in stress tests or by the customer. The problem of testing multi-threaded programs is even more costly because tests that reveal a concurrent fault in the field or in a 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 in recreating the conditions under which it occurred. When the conditions of the bug are finally recreated, the debugging itself may mask the bug (the observer effect).

There is a large body of research involved in trying to improve the quality of multi-threaded programs, both in academic circles and in industry. Progress has been made in many domains and it seems that a high-quality solution must contain components from many of them. Work on race detection [2–8] has been going on for a long time. Race detection tools suffer from the dual problem of having too many false alarms as well as not identifying some of the races. A different approach was taken by a set of tools, called noise makers. Noise makers increase the probability of bugs being discovered by inducing different timing scenarios [9–11]. Noise makers do not do any reporting; instead, they try to make the tests fail. Tools for replay and partial replay [11–13] are necessary for debugging and contain technology that is useful for testing. Static analysis tools of various types [14,15], as well as formal analysis tools, are being developed to detect faults in the multi-threaded domain [16–19]. Analysis tools that show a view of specific interest in the interleaving space both for coverage and performance [20,21] are being worked on. Currently, the most common testing methodology by dollar value takes single thread tests and creates stress tests by cloning them many times and executing them simultaneously [22] (and the commercial tools Rational Robot and Mercury WinRunner). A variety of programming and debugging aids for such an environment are closely related to the testing technologies discussed, but, owing to space limitations, will not be discussed in this paper.

In this paper, we discuss and show initial results for developing a benchmark that formally assesses the quality of different tools and technologies and compares them. Many areas, including some of the technologies discussed in this paper, have benchmarks [23]. The benchmark we are working on is different in that it contains not only programs against which the tools are evaluated, but also a number of additional artifacts that are useful for developing the testing tools. For example, the bugs are annotated so that if a race detection tool suspects a variable, assessment can be made to determine whether it is a false negative or a real result. We have started working on this benchmark, and already created about 40 annotated programs. The benchmark can be seen at http://qp.research.ibm.com/QuickPlace/concurrency_testing/Main.nsf. A password is required and is

available upon request from Shmuel Ur at ur@il.ibm.com. The annotation of bugs includes information about their location in the program, the variables involved, and bug patterns exhibited. The annotation work draws on our ongoing bug taxonomy effort [24].

2.   EXISTING DEDICATED CONCURRENT TESTING TECHNOLOGIES

This section surveys technologies that we think are the most useful or promising for the creation of concurrent testing tools and shows how they could interact. We divide the technologies into two domains. The first includes technologies that statically inspect the program and extract useful information. This information can be in the form of a description of a bug, stating that a synchronization statement is redundant or pointing to missing locks. Static technologies can also be used to generate information that other technologies may find useful, such as a list of program statements from which there can be no thread switch (single thread executing). The static technologies we discuss use formal verification, mainly model checking, and forms of static analysis (the boundary may be blurred [25]). The dynamic technologies are active while the program is executing. The one most commonly associated with concurrent testing is race detection. However, we believe that noise makers, replay, coverage, and performance monitors are also of importance. The technologies described in this paper are white box in that access to the code is assumed.

2.1.   Static techniques

2.1.1.   Formal verification

Model checking is a family of techniques, based on systematic and exhaustive state-space exploration, for verifying the properties of concurrent systems. Properties are typically expressed as invariants (predicates) or formulas in a temporal logic. Model checkers are traditionally used to verify models of software expressed in special modeling languages, which are simpler and higher-level than general-purpose programming languages. Recently, model checkers have been developed that work by directly executing real programs; we classify them as dynamic technologies and discuss under dynamic testing technologies. Manually producing models of software is labor-intensive and error-prone, so a significant amount of research is focused on abstraction techniques for automatically or semi-automatically producing such models. Notable work in this direction includes FeaVer [17], Bandera [19], SLAM [26], Java PathFinder [21], and BLAST [27].

2.1.2.   Static analysis

Static analysis plays two crucial roles in verification and defect detection. First, it is the foundation for constructing models for verification, as described above. *Dependency analysis*, in the form of *slicing*, is used to eliminate parts of the program that are irrelevant to the properties of interest. *Pointer analysis* is used to conservatively determine which locations may be updated by each program statement. This information is then used to determine the possible effects of each program statement on the state of the model. For concurrent programs, *escape analysis*, such as [14], is used to determine which variables

are thread-local and which can be shared. This information can be used to optimize the model or to guide the placement of instrumentation used by dynamic testing techniques.

Second, static analysis can be used by itself for verification and defect detection. Compared with model checking, static analysis is typically more scalable but more likely to give indeterminate results ('don't know'). One approach is to develop specialized static analysis for verifying specific concurrency-related properties. For example, there are specialized type systems [6,28,29] and data-flow analyzers [30,31] for detecting data races; some of these analyses, specifically [28] and [30], also detect deadlocks. The type systems are modular, scalable, and conservative (i.e. they never overlook errors). However, they require programmers to provide annotations in the program, and they produce false alarms if the program design is inconsistent with the design patterns encoded in the type system. Static conflict analysis [31] is conservative and automatic, but less scalable than the type systems. RacerX [30] is scalable and automatic but not conservative (it can miss some errors). There are also general verification-oriented static analysis frameworks, such as Canvas [32], ESP [33], and xgcc [34], but these generally do not model concurrency and are potentially unsound when applied to concurrent programs. Nevertheless, in practice, they may still be effective at finding some concurrency-related errors (e.g. forgetting to release a lock [34]). TVLA [35] is a general static analysis framework that can model concurrency and rigorously verify a variety of properties for concurrent programs, as demonstrated in [36,37]. However, TVLA's analysis is relatively expensive and, hence, limited to small programs.

2.2.   Dynamic testing technologies

All the dynamic testing technologies discussed in this section make use of instrumentation technology. An instrumentor is a tool that receives as input the original program (source or object) and instruments it, at different locations, with additional statements. During the execution of the program, the instructions embedded by the instrumentor are executed.

In the following, it is assumed that an instrumentor is available. The benchmarks include a paper [38] and documentation including code on how to easily use AspectJ [39] for the instrumention needed for the technologies discussed.

2.2.1.   *Noise makers*

A noise maker [9–11] belongs to the class of testing tools that make tests more likely to fail and thus increase the efficiency of testing. In the sequential domain, such tools [40,41] usually work by denying the application certain services, for example, indicating that no more memory is available to a memory allocation request. In the sequential domain, this technique is very useful but is limited to verifying that, on the bad path of the test, the program fails gracefully. In the concurrent domain, noise makers are tools that force different legal interleavings for each execution of the test in order to check that the test continues to perform correctly. In a sense, it simulates the behavior of other possible schedulers. During the execution of the program, the noise heuristic receives calls embedded by the instrumentor. When such a call is received, the noise heuristic decides, randomly or based on specific statistics or coverage, if some kind of delay is needed. Two noise makers can be compared with each other with regard to the performance overhead and the likelihood of uncovering bugs.

2.2.2. Race and deadlock detection

A race is defined as accesses to a variable by two threads, at least one of which is a write, which have no synchronization statement temporally between them [3]. A race is considered an indication of a bug. Race detectors are tools that look, on-line or off-line, for evidence of existing races. Typically, race detectors work by first instrumenting the code, such that the information will be collected; then they process the information. On-line race detection suffers from performance problems and tends to significantly slow down the application. On-line race detection techniques compete in the performance overhead they produce. Off-line race detection sometimes suffers from the fact that huge traces may be produced, and compete in reducing and compressing the information needed. The main problems of race detectors of all types are that they produce too many false alarms and that they are susceptible to scheduling order.

A deadlock is defined as a state where, in a collection of threads, each thread tries to acquire a lock already held by one of the other threads in the collection. Hence, the threads block each other in a cyclic manner. Tools exist that can examine traces for evidence of deadlock potentials [42,43]. Specifically, they look for cycles in lock graphs.

2.2.3. Replay

One of the most annoying features of concurrent testing is that once a bug is detected, it may be very difficult to debug. There are two distinct aspects to this problem. The first is that often the bug does not reproduce with a high enough probability. The second is that even if it does, when you try to analyze it using a debugger or print statements, it does not manifest. The ability to replay a test is essential for debugging [13,44,45]. Replay has two phases: record and playback. In the record phase, information concerning the timing and any other 'random' decision of the program is recorded. In the playback phase, the test is executed and the replay mechanism ensures that the same decisions are taken. Doing full replay [12] may be difficult and may require the recording of large amounts of information. Partial replay, which causes the program to behave as if the scheduler is deterministic and repeats the previous test [11], is much easier and, in many cases, good enough. Partial replay algorithms can be compared on the likelihood of performing correct replay and on their performance.

2.2.4. Coverage

Malaiya *et al.* [46] showed a correlation between good coverage and high-quality testing, mainly at the unit level. The premise, albeit simplified, is that it is very useful to check that we have gone through every statement. This coverage measure is of very little utility in the multi-threading domain. The most promising avenue for creating multi-threaded coverage models is to create models that cover bug patterns. For example, checking that variables on which contention can occur, had contention in the testing (ensuring possible races). Another concrete example is synchronization coverage, a model with a task for each synchronization, which checks that this synchronization statement had impact on the flow of the test. A synchronization is utilized if it either stopped another thread or was stopped by it. These two coverage models are implemented in ConTest [11]. Additional coverage measures should be created and their correlation to bug detection studied.
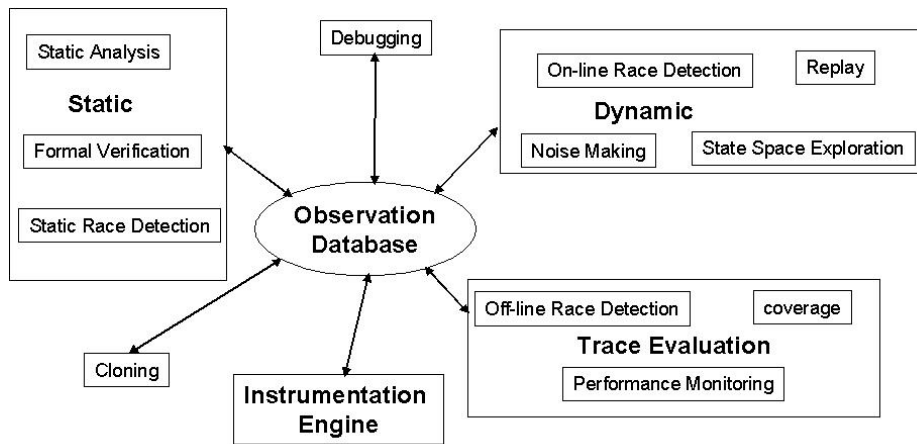
Figure 1. Interrelations between technologies.

2.2.5. *Systematic state-space exploration*

Systematic state-space exploration [16–19,21,47,48] is a technology that integrates automatic test generation, execution, and evaluation in a single tool. The idea is to systematically explore the state spaces of systems composed of several concurrent components. Such tools systematically explore the state space of a system by controlling and observing the execution of all the components, and by reinitializing their executions. They typically search for deadlocks, and for violations of user-specified assertions. Whenever an error is detected during state-space exploration, a scenario leading to the error state is saved. Scenarios can be executed and replayed. To implement this technology, replay technology is needed to force interleavings, instrumentation is needed, and coverage is advisable, so the tester can make informed decisions on the progress of the testing.

3.   INTERACTIONS BETWEEN TECHNOLOGIES

Figure 1 illustrates a high-level depiction of a suggested design as to how the different technologies can interact. Different technologies talk to each other through the observation database. Instrumentation is an enabling technology for all the technologies included in the dynamic and trace evaluation boxes. Some technologies are orthogonal and have no awareness that another technology is being used. For example, coverage can be measured for cloned tests. In such a case, the two technologies do not have to share anything through the observation database. This section uses examples to demonstrate ways in which technologies can be combined to yield additional value.

The observation database contains the following information (partial suggestion).

- Interesting variables—for example, variables that could be involved in races or bugs.
- Possible race locations—locations in the programs that are suspect.
- Unimportant locations—areas that are well synchronized, for example, areas in which only one thread is alive.
- Coverage information—database showing which coverage tasks [49] were covered.
- Traces of executions—to be used by off-line analyzers.

Instrumentation is the process of automatically modifying code by adding user exits. The instrumentor is told what to instrument by the observation database. Input to the instrumentor may include which parts of code (e.g. files, classes, methods, lines), which subset of the variables, where to instrument, and what to put at each point. A natural selection, which already has most of the required features, is AspectJ [39]. Augmention of AspectJ so it can do all the work required from an instrumentor in this framework was studied in [38] and is not a very difficult job.

The static technologies (static analysis and formal verification) can be used directly for finding bugs, and can also be used to create information that is useful for other technologies. The work by Choi *et al.* [50,51] and von Praun and Gross [31] are good examples of the use of static analysis to optimize run-time race detection. With the observation database, one may improve a race detector by using an existing static analyzer.

The information gleaned in static analysis can also be transferred to the instrumentor and used to reduce the overhead (i.e. instrument only what really needs instrumenting), or by attaching information that is used in run-time to the instrumentation calls. For example, one of Stoller's improvements [9] when compared with ConTest was to use static analysis to find locations that do not need instrumentation. Had the suggested architecture been used to implement Stoller's idea, a small modification would have been necessary, without writing a project from scratch.

The technologies are already combined in a variety of ways. For example, ConTest contains an instrumentor, a noise generator, a replay and coverage component, and listenter architecture on top of which a deadlock discipline violation was created and race detectors are planned in the near future. The integration of the components is integral to the service that ConTest gives to testing multi-threaded Java programs. With ConTest, tests can be executed multiple times to increase the likelihood of finding bugs (instrumentor and noise). Once a bug is found, replay is used to debug it. Coverage is used to evaluate the quality of the testing, and static analysis improves the coverage quality and the performance of the noise maker. Another example is Java PathExplorer (JPaX) [43,52], a run-time monitoring tool for monitoring the execution of Java programs. It automatically instruments the Java bytecode of the program to be monitored and inserts logging instructions. The logging instructions write events relevant for the monitoring to a log file, or to a socket, in the case that on-line monitoring is requested. Event traces are examined for data races (using the Eraser algorithm) and deadlock potentials. Furthermore, JPaX can monitor that an execution trace conforms with a set of user provided properties stated in temporal logic.

Such technologies currently require mastery of a number of different technologies. One of the goals of this proposed project is to create a standard interface between technologies and the observation database, so that improvement in one tool can be used to improve the overall solution. The assumption is that a good solution will have a number of components. It is important that a researcher can work on one component, use the rest 'off-the shelf', and check the global impact of her work.

4. BENCHMARK

The different technologies for concurrent testing can be compared to each other based on: the number of bugs they can find or the probability of finding bugs, the percentage of false alarms, and performance overhead. Sometimes the technology itself cannot be compared, as it is only part of a solution and the improvement in the solution, as a whole, must be evaluated. To facilitate the comparison, we are creating a benchmark that is composed of the following two parts.

1. A repository of annotated programs which can be used to evaluate technologies. This repository is already in use by a number of researchers.
2. An architecture containing supplied components that help in developing testing tools.

The repository contains programs on which the technologies can be evaluated. Each program comes with artifacts such as:

- source code (and instrumented bytecode) in standard project format;
- test cases and test drivers;
- documentation of the bugs in each program;
- sample traces of program executions (not implemented yet).

The repository of programs includes many small programs as well as large programs that illustrate specific bugs from the field. The programs with the bugs and the instrumented programs are already available and the traces will be generated on demand. This makes evaluating many of the technologies much easier.

The second component of the benchmark is a repository of tools, together with the observation database. This way, researchers can use a mix-and-match approach and complement their components with benchmark components, to create and evaluate solutions based on the created whole. The components include an instrumentor, which is needed in most solutions, as well as noise makers and race detection components.

4.1. Experience gathered in starting the benchmark

In an effort to start composing the benchmark, we asked students of an undergraduate software testing class to write benchmark programs containing one (or more) concurrent bugs. As programs created by the students are biased toward bugs typical of novice programmers, this was just a beginning. The assignment can be seen at http://cs.haifa.ac.il/courses/softtest/testing2003.

By using raceFinder [10] to produce test reports, and working with a large number of different users, we were able to reason about raceFinder and noise making in general. We learned, by looking at the raceFinder heuristics, that creating noise in a few selected locations is more effective than noise everywhere for finding concurrent bugs. It is advisable to use static analysis [14,36], dynamic analysis [53,54], or both [50] to identify the best program locations. Testing many programs with non-atomic bug patterns supports the claim that raceFinder, and by extension noise makers, can effectively handle this type of bug pattern and uncover the programs' non-atomic buggy interleavings. Atomicity is a common higher-level correctness requirement that expresses non-interference between concurrently executed code blocks. A code block is atomic if every execution of the program is equivalent to an

execution in which that code block is executed without being interleaved with actions of other threads. The non-atomic bug patterns happen when the programmer assumes that something is atomic and it is not.

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 manifestation (or no manifestation) of concurrent bugs. The three main factors identified were as follows.

- Scheduling policy of the Java Virtual Machine (JVM)—this policy is usually deterministic and thus produces a limited subset of the interleaving space [12,55].
- Input of the program (control flow)—some of the interleavings that induce a concurrent bug are input-dependent.
- Design of the program—a problem arises when the design contains a fault (not the implementation) and this fault manifests itself only in rare circumstances, requiring complex scenarios.

Another less common factor is the two levels of the Java memory model. 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 (e.g. the JVM definition of the order in which the waiting threads are awakened by a notify) cannot be impacted. In addition, bugs related to two-tier memory hierarchy, will not be observed on a one-tier memory Java implementation, regardless of the scheduling.

Each of the identified factors relates to a different set of reasons that lead to a bug manifesting. Ultimately, we would like to handle all of these aspects while testing in order to increase the likelihood that the bug does appear; however, many testing tools deal only with one single aspect. For example, intelligent noise makers can effectively change JVM scheduling to manifest concurrent bugs; however, they cannot control the input. Thus, good testing input metrics are required when the bugs are also related to specific input or inputs. There are tools designed to give coverage of the program's logic (see ConAn [56]), and it could be interesting to combine a noise maker with such 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 students' non-standard programming practices, which were not considered in the tool design.

The benchmark website contains a table summarizing important aspects of the benchmark suite. The table is updated as the benchmark grows. Thus, the updated table is maintained on the benchmark Web site.

Here we briefly summarize the current status. The benchmark currently contains 40 programs with a range of sizes. Currently there are seven programs under 100 lines, 20 between 100 and 300 lines, six between 300 and 500 lines, three between 500 and 1000, three between 1000 and 3000, and one program larger than 15 000 lines. The programs contain bugs from 12 categories [7,24,57]: not atomic, orphaned thread, deadlock, sleep, double-checked locking, notify instead of notifyAll, Blocking-Critical-Section, Missing Condition-For-Wait, Unguarded if statement, High-level data race, Non-atomic floating point operation, and General race. Approximately two-thirds of the bugs are in the not-atomic category (due to data being accessed without a lock or when holding the wrong lock). Deadlock is the second most common type of bug, and the remaining bugs are scattered among the

remaining categories. Although the benchmark is not balanced between different bug types, it provides broad coverage of the concurrent bug types known in the literature. One of the bugs comes from a released commercial product, others from open-source code obtained on the Web, about a third from programs created by tool developers, and the rest from programs created by students specifically for the benchmark. The students' programs contain novice-level bugs. The bugs in the commercial and open-source programs are more subtle and seem to be representative of errors made by experienced programmers. The bugs in the programs created by tool developers range from simple to subtle.

5.   CONCLUSIONS

In this paper, we discussed the problem of evaluating multi-threaded testing technology and creating a benchmark that will enable research in this domain. There are many technologies involved and improvements in the use of one technology may depend on utilizing another. We believe that greater impact, and better tools, could result if use were made of a variety of relevant technologies. Towards this end, we would like to start an enabling project that will help create useful technologies, evaluate them, and share knowledge. There are specific attempts at creating tools that are composed of a variety of technologies [1,11], but they do not provide an open interface for extension and do not support the evaluation of competing tools and technologies.

The suggested framework is an ideal tool to be used in education. The amount of code needed to build a coverage, noise, race detection, or replay tool is a few hundred lines of code and is easily within the scope of a class exercise. Indeed, this was one of the motivations of this paper, as the work reported in [10] started this way.

We discussed this project at PADTAD 2003 and PADTAD 2004 and with additional groups such as the AspectJ developers. Quite a few groups and researchers have expressed interest in participating in this project. We are looking at formal structures under which this project could be held.

In the direction of the benchmark, we have made some slow progress since suggesting the project in April 2003. We gave our undergraduate software testing class students an assignment to write programs containing one (or more) concurrent bugs. In testing the homework assignments we found some bugs in our tools, mainly because the students programmed in ways we had never 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 automate the detection. 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 towards the kind of bugs that novice programmers create.

A good source for bugs created by experienced programmers is the open-source code. One way to collect such bugs for the benchmark is to follow the bug fix errata and ask the owners for the source code containing the bug. This way we can have the bug and the correct fix for the benchmark.

We saw a number of bugs that could not 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. Bugs that have to cope with scheduling inside the JVM cannot be impacted, for example, the JVM definition of the order in which the waiting threads are awakened by a notify. 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 single 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 learned about creating benchmarks in general, and creating benchmarks using student assignments in particular. As a result of our preliminary use of the benchmark, we also have a better idea of how to further expand the benchmark. This is an ongoing work in which we are expanding the benchmark and adding more features. In the near future, we expect to get additional feedback from benchmark users, which we will use in the next iterations of this exercise.

REFERENCES

1. Cunha JC, Kacsuk P, Winter SC (eds.). *Parallel Program Development For Cluster Computing*. Nova Science: New York, 2000.
2. Richards B, Larus JR. Protocol-based data-race detection. *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*. ACM Press: New York, 1998; 40–47.
3. Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems* 1997; **15**(4):391–411.
4. Itzkovitz A, Schuster A, Zeev-Ben-Mordehai O. Toward integration of data race detection in dsm systems. *Journal of Parallel and Distributed Computing* 1999; **59**(2):180–203.
5. Netzer R, Miller B. Detecting data races in parallel program executions. *Proceedings of the 1990 Workshop on Advances in Languages and Compilers for Parallel Computing*, Irvine, CA, 1990. MIT Press: Cambridge, MA, 1990; 109–129.
6. Flanagan C, Freund SN. Detecting race conditions in large programs. *Proceedings of the Program Analysis for Software Tools and Engineering Conference*, June 2001. ACM Press: New York, 2001.
7. Artho C, Havelund K, Biere A. High-level data races. *Proceedings of the 1st International Workshop on Verification and Validation of Enterprise Information Systems (VVEIS'03)*, Angers, France, 2003. ICEIS Press, 2003.
8. Artho C, Havelund K, Biere A. Using block-local atomicity to detect stale-value concurrency errors. *Proceedings of the 2nd International Symposium on Automated Technology for Verification and Analysis (AVTA)* (*Lecture Notes in Computer Science*, vol. 3299). Springer: Berlin, 2004; 150–164.
9. Stoller SD. Testing concurrent Java programs using randomized scheduling. *Proceedings of the 2nd Workshop on Runtime Verification (RV)* (*Electronic Notes in Theoretical Computer Science*, vol. 70(4)). Elsevier: Amsterdam, 2002.
10. Ben-Asher Y, Eytani Y, Farchi E. Heuristics for finding concurrent bugs. *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2003) PADTAD Workshop*, April 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003.
11. Edelstein O, Farchi E, Nir Y, Ratsaby G, Ur S. Multithreaded Java program test generation. *IBM Systems Journal* 2002; **41**(1):111–125. Available at: http://www.research.ibm.com/journal/sj/411/edelstein.html.
12. Choi J-D, Srinivasan H. Deterministic replay of Java multithreaded applications. *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, Welches, OR, August 1998. ACM Press: New York, 1998; 48–59.
13. Schuppan V, Baur M, Biere A. JVM independent replay in Java. *Proceedings of the 4th Workshop on Runtime Verification (RV2004)*, Barcelona, 3 April 2004; 76–94. Available at: http://citeseer.ist.psu.edu/schuppan04jvm.html.
14. Choi J-D, Gupta M, Serrano M, Sreedhar V, Midkiff S. Escape analysis for Java. *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press: New York, 1999.
15. Hovemeyer D, Pugh W. Finding bugs is easy. *OOPSLA Companion*. ACM Press: New York, 2004; 132–136.
16. Visser W, Havelund K, Brat G, Park S, Lerda F. Model checking programs. *Automated Software Engineering Journal* 2003; **10**(2):203–232.
17. Holzmann GJ, Smith MH. Software model checking: Extracting verification models from source code. *Proceedings of the International Conference on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV)*. Kluwer: Dordrecht, 1999; 481–497.
18. Stoller SD. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer* 2002; **4**(1):71–91.
19. Corbett JC, Dwyer M, Hatcliff J, Pasareanu C, Robby S, Laubach S, Zheng H. Bandera: Extracting finite-state models from Java source code. *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*. ACM Press: New York, 2000.

20. Souter A, Yang C-S, Pollock L. All-du-path coverage for parallel programs. *Proceedings of the ACM SigSoft International Symposium on Software Testing and Analysis. ACM SIGSOFT* 1998; **23**(2):153–162.

21. Havelund K, Pressburger T. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer* 2000; **2**(4):366–381.

22. Hartman A, Kirshin A, Nagin K. A test execution environment running abstract tests for distributed software. *Proceedings of Software Engineering and Applications (SEA 2002)*, 2002. ACTA Press: Anaheim, CA, 2002.

23. Avrunin GS, Corbett JC, Dwyer MB, Pasareanu CS, Siegel SF. Comparing finite-state verification techniques for concurrent software. *Technical Report UM-CS-1999-069*, Department of Computer Science, University of Massachusetts at Amherst, MA, 1999.

24. Nir Y, Farchi E, Ur S. Concurrent bug patterns and how to test them. *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2003) PADTAD Workshop*, April 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003.

25. Rustan K, Leino M. *Extended Static Checking: A Ten-year Perspective* (*Lecture Notes in Computer Science*, vol. 2000). Springer: Berlin, 2001.

26. Ball T, Majumdar R, Millstein T, Rajamani SK. Automatic predicate abstraction of C programs. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press: New York, 2001; 203–213.

27. Henzinger TA, Jhala R, Majumdar R, Sutre G. Lazy abstraction. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press: New York, 2002; 58–70.

28. Boyapati C, Lee R, Rinard M. Ownership types for safe programming: Preventing data races and deadlocks. *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press: New York, 2002; 211–230.

29. Grossman D. Type-safe multithreading in Cyclone. *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*. ACM Press: New York, 2003; 13–25.

30. Engler DR, Ashcraft K. RacerX: Effective, static detection of race conditions and deadlocks. *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP)*. ACM Press: New York, 2003; 237–252.

31. von Praun C, Gross T. Static conflict analysis for multi-threaded object-oriented programs. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press: New York, 2003; 115–128.

32. Ramalingam G, Warshavsky A, Field J, Goyal D, Sagiv M. Deriving specialized program analyses for certifying component-client conformance. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press: New York, 2002; 83–94.

33. Das M, Lerner S, Seigle M. ESP: Path-sensitive program verification in polynomial time. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press: New York, 2002; 57–68.

34. Hallem S, Chelf B, Xie Y, Engler D. A system and language for building system-specific, static analyses. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press: New York, 2002; 69–82.

35. Lev-Ami T, Reps T, Sagiv M, Wilhelm R. Putting static analysis to work for verification: A case study. *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press: New York, 2000; 26–38.

36. Yahav E. Verifying safety properties of concurrent Java programs using 3-valued logic. *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*. ACM Press: New York, 2001; 27–40.

37. Yahav E, Sagiv M. *Automatically Verifying Concurrent Queue Algorithms* (*Electronic Notes in Theoretical Computer Science*, vol. 89). Elsevier: Amsterdam, 2003.

38. Copty S, Ur S. Multi-threaded testing with AOP is easy, and it finds bugs! *Proceedings of Euro-Par*, 2005; 740–749. DOI: 10.1007/11549468-81.

39. Schmidmeier A, Hanenberg S, Unland R. Implementing known concepts in AspectJ. http://citeseer.ist.psu.edu/571060.html.

40. Miller B, Koski D, Lee CP, Maganty V, Murthy R, Natarajan A, Steidl J. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. *Technical Report*, University of Wisconsin, Madison, WI, 1995.

41. Whittaker JA. *How to Break Software*. Addison-Wesley: Reading, MA, 2003.

42. Harrow JJ. Runtime checking of multithreaded applications with visual threads. *SPIN Model Checking and Software Verification* (*Lecture Notes in Computer Science*, vol. 1885), Havelund K, Penix J, Visser W (eds.). Springer: Berlin, 2000; 331–342.

43. Havelund K, Rosu G. Monitoring Java programs with Java PathExplorer. *Proceedings of the 1st Workshop on Runtime Verification (RV'01)*, Paris, France, 23 July 2001 (*Electronic Notes in Theoretical Computer Science*, vol. 55(2)). Elsevier: Amsterdam, 2001.

44. Choi J-D, Srinivasan H. Deterministic replay of Java multithreaded applications. *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*. ACM Press: New York, 1998.

45. Srinivasan H, Konuru R, Choi J-D. Deterministic replay of distributed Java applications. *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, 2000.
46. Malaiya Y, Li N, Bieman J, Karcich R, Skibbe B. Software test coverage and reliability. *Technical Report*, Colorado State University, 1996.
47. Godefroid P. Model checking for programming languages using Verisoft. *Proceedings of the Symposium on Principles of Programming Languages*. ACM Press: New York, 1997; 174–186.
48. Musuvathi M, Park D, Chou A, Engler DR, Dill DL. CMC: A pragmatic approach to model checking real code. *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002. USENIX Association: Berkeley, CA, 2002.
49. Ratsaby G, Sterin B, Ur S. Improvements in coverability analysis. *Proceedings of the 11th Formal Methods Europe International Symposium (FME 2002)*. Springer: Berlin, 2002; 41–56.
50. Choi J-D, Lee K, Loginov A, O'Callahan R, Sarkar V, Sridharan M. Efficient and precise datarace detection for multithreaded object-oriented programs. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press: New York, 2002; 258–269.
51. O'Callahan R, Choi J-D. Hybrid dynamic data race detection. *Proceedings of the ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press: New York, 2003.
52. Barringer H, Goldberg A, Havelund K, Sen K. Rule-based runtime verification. *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2004)*, Venice, January 2004 (*Lecture Notes in Computer Science*, vol. 2937). Springer: Berlin, 2004; 44–57.
53. Flanagan C, Freund SN. Atomizer: A dynamic atomicity checker for multithreaded programs. *Proceedings of the 31st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM Press: New York, 2004.
54. Wang L, Stoller SD. Run-time analysis for atomicity. *Proceedings of the 3rd Workshop on Runtime Verification (RV)* (*Electronic Notes in Theoretical Computer Science*, vol. 89(2)). Elsevier: Amsterdam, 2003.
55. Edelstein O, Farchi E, Goldin E, Nir Y, Ratsaby G, Ur S. Testing multi-threaded Java programs. *IBM System Journal (Special Issue on Software Testing)* 2002; **41**(1):111–125.
56. Long B, Hoffman D, Strooper P. Tool support for testing concurrent Java components. *IEEE Transactions on Software Engineering* 2003; **29**(6):555–566.
57. Long B, Strooper P. A classification of concurrency failures in Java components. *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*. IEEE Computer Society Press: Los Alamitos, CA, 2003; 287.1.