# Benchmark and Framework for Encouraging Research on Multi-Threaded Testing Tools

Klaus Havelund
Kestrel Technology
NASA Ames Research Center
Moffett Field, CA 94035-1000 USA
havelund@email.arc.nasa.gov

Scott D. Stoller
Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794, USA
stoller@cs.sunysb.edu

Shmuel Ur
IBM Haifa Research Lab
Haifa University Campus
Haifa, 31905, Israel
ur@il.ibm.com

## Abstract

*A problem that has been getting prominence in testing is that of looking for intermittent bugs. Multi-threaded code is becoming very common, mostly on the server side. As there is no silver bullet solution, research focuses on a variety of partial solutions. In this paper (invited by PADTAD 2003) we outline a proposed project to facilitate research. The project goals are as follows. The first goal is to create a benchmark that can be used to evaluate different solutions. The benchmark, apart from containing programs with documented bugs, will include other artifacts, such as traces, that are useful for evaluating some of the technologies. The second goal is to create a set of tools with open API's that can be used to check ideas without building a large system. For example an instrumentor will be available, that could be used to test temporal noise making heuristics. The third goal is 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, could congregate.*

## 1. Introduction

The increasing popularity of concurrent Java 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. The development of technology and tools for identifying concurrent defects are now considered by some experts in the domain as the most important issue that needs to be addressed in software testing [11].

There are a number of distinguishing factors between concurrent defect analysis and sequential testing and these differences make it especially challenging. One problem 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 under the simple conditions of unit testing the scheduler is deterministic. Because of this, executing the same tests repeatedly does not help. Due to this fact, concurrent bugs are often not found early in the process but rather only in stress test 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).

All the currently used testing techniques, some of which, such as coverage and inspection, proved very useful, address sequential problems. A solution that is being attempted is to hide the multi-threading from the user [1]. However, no architecture has been found that lets the user take full advantage of the fact that the program is multi-threaded or that the hardware is parallel and yet lets her program as if intermittent bugs were not a problem. Existing

attempts only serve to hide the bugs even further, because the programmer is not aware that she can cause such bugs.

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 solution of high quality will contain components from many of them. Work on race detection [29] [30] [22] [27] [13] has been going on for a long time. Race detection suffers from the problem of false warnings. To alleviate this problem, tools have been developed which try to increase the probability of bugs being discovered by creating more races have been developed [32] [12]. The tools that cause races do not report any false alarms (they actually do not report anything), they just try to make the user tests fail. Tools for replay [9], necessary for debugging and containing technology useful for testing, have been developed. It is hard to create replay that always work. Therefore, tools that increase the probability of replay have also been developed [12]. Static analysis tools of various types, as well as formal analysis tools, are being developed, which can detect faults in the multi-threaded domain [31] [18] [10]. Analysis tools that show a view of specific interest in the interleaving space both for coverage and performance [6][18] are being worked on. Currently the most common testing methodology by dollar value is taking single thread tests and creating stress tests by cloning them many times and executing them simultaneously [17] (Rational Robot and Mercury WinRunner). There are a number of approaches to cloning, some of which are very practical. In addition, outside the scope of this work but closely related, are a variety of programming and debugging aids for such an environment.

There is a need for a benchmark for formally assessing the quality of different tools and technologies and for comparing them. Many areas, including some of the technologies discussed in this paper have benchmarks [2]. What we suggest in this paper is different in that we not only help compare the tools but also build the tools. This is done in a number of ways: artifacts that are useful for testing are available in addition to the test programs; and a framework of tools is available so that the user need only change his component. Not only tools, but also research in areas such as bug classification, can be helped by this process.

Section 2 lists the technologies which we think are relevant to the benchmark. Section 3 details the interaction between the technologies. Section 4 explains what the benchmark will contain, and we conclude in section 5.

## 2. Existing dedicated concurrent testing technologies

In this section we survey some technologies which we think are the most useful or promising for the creation of concurrent testing tools and show how they could interact. We divide the technologies into two domains. The first includes technologies that statically inspect the program and glean some information. This information could 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. The static technologies which we discuss are formal verification mainly model checking and forms of static analysis. The second group of 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. A third group which we mention but not discuss is trace analysis technologies. Some technologies such as race detection, coverage or performance monitoring can be performed on-line and off-line. The trade off is usually that on-line affects performance and off-line requires huge storage space. As the underlying technologies is very similar we will mostly discuss the on-line version in the section on dynamic technologies. In addition, we talk about cloning, which is currently the most popular testing technique for concurrent bugs in the industry. The technologies described in this paper are white box in that knowledge of the code is assumed. However, cloning is a black box technique, usually deployed very late in the development process. Cloning is mentioned here principally for completeness.

### 2.1. Static testing techniques

*Formal Verification* - Model checking is a family of techniques, based on systematic and exhaustive state-space exploration, for verifying 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 them in section 2.2.) Producing models of software manually is labor-intensive and error-prone, so a significant amount of research is focused on abstraction techniques for producing such models automatically or semi-automatically. Notable work in this direction includes FeaVer [21], Bandera [10], SLAM [3], and BLAST [20].

Model checking is infeasible for concurrent systems with very large state spaces, so the goal is not only to translate the program into the modeling language, but also to determine which details of the program are not essential for ver-

ifying the required properties and to omit those details from the model. The models should normally be *conservative*: they may over-approximate, but not under-approximate, the possible behaviors of the program. Thus, if a conservative model of a program satisfies a given invariant, then so does the program.

*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 [7], is used to determine which variables are thread-local and which may 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 to model checking, program analysis is typically more scalable but more likely to give indeterminate results ("don't know"). One approach is to develop specialized static analyses for verifying specific properties. For example, there are type systems for detecting data races and deadlocks [13] [5]. These type systems are modular and scalable, but 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. There are also static analysis frameworks that can handle large classes of properties. Frameworks that use only conservative analyses, such as TVLA [23] and Canvas [28], support both verification and defect detection. TVLA can conservatively analyze systems with an unspecified and unbounded number of threads. Engler's analysis framework [15] is not conservative (it might miss some violations of the specified properties) and does not deal with concurrency directly. However, it has demonstrated effectiveness at finding bugs in real operating system code, including some concurrency-related bugs, e.g., forgetting to release a lock.

### 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. The instrumentor should have a standard interface that let the user tell it what type of instructions to instrument, which variables, and where to instrument in terms of methods and classes. In addition, the same interface tells it what code to insert in these locations. This interface enables the user of the instrumentor (be it noise maker, race analyzer, replay or coverage tool) to take full advantage of the tool. It also enables performance enhancements, such as not instrumenting in locations where static analysis shows instrumentation to be unnecessary.

The instrumentation can be at the source code, the byte code or the JVM level. The JVM level has the benefit of being the easiest but is the least transportable. Both the bytecode and the source are transportable. Instrumenting at the bytecode level is easier and is therfore the most common.

In the following, it is assumed that an instrumentor is available.

*Noise makers* - A noise maker [12] [32] 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 [25] [33] usually work by denying the application some services, for example returning 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 performs correctly. In a sense, it simulates the behaviour of other possible schedulers. The noise heuristic, during the execution of the program, 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 to each other with regard to the performance overhead and the likelihood of uncovering bugs.

There are two important research questions in this domain. The first to find noise making heuristics with a higher likelihood of uncovering bugs. The second, important mainly for performance but also for the likelihood of finding bugs, is the question of where calls to the heuristic should be embedded in the original program.

*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 [30]. A race is considered an indication of a bug. Race detectors are tools that look, online or offline, for evidence of existing races. Typically, race detectors work by first instrumenting the code such that the information will be collected and then they process it. On-line race detection suffers from performance problems and tends to slow down the application significantly. On-line race detection techniques compete in the performance overhead they produce. Off-line race detection suffers from the fact that huge traces are produced, and techniques compete

in reducing and compressing the information needed. The main problem of race detectors of all breeds is that they produce too many false alarms.

While the definition of race used by the tools is similar, the ability to detect user implemented synchronization is different. Detecting such synchronization with high probability will alleviate much of the problem of false alarms.

Annotated traces of program executions can help race detection research. The trace will include all the information needed by the race detection tools, such as memory location accessed and synchronization events. In addition, for each record, annotated information is kept about why it was recorded, so that the race detection tool can decide if it should consider this record. The annotation will also denote the bugs revealed by the trace so that the ratio between real bugs and false warnings can be easily verified.

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 are blocked on each other in a cyclic manner. Tools exists which can examine traces for evidence of deadlock potentials [16] [19]. Specifically they look for cycles in lock graphs.

*Replay* - One of the most annoying features of concurrent testing is that once a bug is seen it may be very difficult to remove. There are two distinct aspects of this problem. The first is that many times the bug does not reproduce with high enough probability. The second is that even if it does, when you try to analyze it using a debugger or print statements, it goes away. The ability to replay a test is essential for debugging. 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. There are many possible sources of randomness in the execution of a test on a program. Some apply even to sequential algorithm, for example, the most obvious are random and time functions. Less obvious sources might include using a hash function where the order of the objects taken out depends on the location in memory and varies from execution to execution. Another source of randomness is the thread scheduler, which can choose a different location for the context switches in different executions. Doing full replay [9] may be difficult and may require the recording of a lot of information as well as wrapping many functions. Partial replay, which causes the program to behave as if the scheduler is deterministic and repeats the previous test [12] is much easier and, in many cases, good enough. Partial replay algorithms can be compared on the likelihood of performing replay and on their performance. The latter is significant in the record phase overhead, and not so much in the replay phase.

*Coverage* - Malaiya et al [24] 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 multithreading domain. An equivalent process, in the multithreaded domain, is to check that variables on which contention can occur had contention in the testing. Such measures exist in ConTest [12]. Better measures should be created and their correlation to bug detection studied. A new and interesting research question is to use coverage in order to decide, given limited resources, how many times each test should be executed. The reason a test should be executed more than once is that even if the test can potentially find a bug in the concurrent domain, it is not guaranteed, or even likely, to do so.

All the concurrent coverage models that have been created suffer from the problem that most tasks are not feasible. For example, consder the model: for all variables, a variable is covered if it has been touched by two threads. Most tasks in this model are not feasible. Static techniques could be used to evaluate which variables can be accessed by multiple threads. This evaluation is needed to create the coverage metric.

*Systematic state space exploration* - Systematic state space exploration [14] [31] [18] 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 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 that the tester can make informed decisions. on the progress of the testing. Another systematic state-space exploration tool, for C programs, is CMC [26]. Unlike VeriSoft, CMC uses traditional state-based search algorithms, not state-less search, so it uses "clone" procedures to copy the system state, and does not rely on replay.

## 2.3. Cloning

Cloning, also called load testing, is the most commonly used testing technique aimed at finding intermittent bugs and evaluating performance. Cloning is used at the tail end of development, either at system testing or as a specialized phase called stress testing. The idea, used in common commercial tools for testing client server applications such as

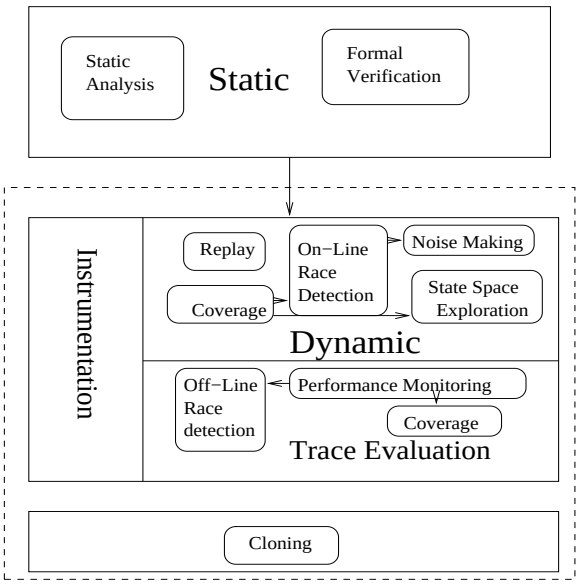


**Figure 1. Interrelations between technologies**

Rational Robot or Mercury LoadRunner, is to take sequential tests and clone them many times. This technique has the advantage of being both relatively simple and very efficient. Because the same test is cloned many times, contentions are almost guaranteed. There are a number of problems, which require careful design. The first is that the expected results of each clone need to be interpreted, so verifying if the test passed or failed is not necessarily straightforward. Many times, changes that distinguish between the clones are necessary. This technique is purely a black box technique. It may be coupled with some of the techniques suggested above, such as noise making or coverage, for greater efficiency.

## 3. Interactions between technologies

Figure 1 contains a high level depiction of the interaction between the different technologies. Each technology is contained in a rounded box. The edges represent the flow of information. For example, information produced by static analysis can be used by the dynamic and trace evaluation technologies. Instrumentation is an enabling technology for all thei technologies included in the dynamic and trace evaluationi boxes. The dashed line around the cloning and the various dynamic analysis techniques signifies that there is value in using the techniques at the same time; however, no integration is needed. The technologies are orthogonal and there is even no awareness that the other technology is being used. For example, coverage can be measured for cloned tests.

The static technologies (static analysis and formal verification), besides being used directly for finding bugs, can be used to create information that is useful for other technologies. Choi et al's work [8] is a nice example of the use of static analysis to optimize run-time race detection. Instrumentation is an enabling technology that is a required part of many technologies. The instrumentation technology needed for all the dynamic technologies and for off-line race detection is virtually identical.

Technologies may be combined in a variety of ways. For example, ConTest contains an instrumentor, a noise generator, a replay and coverage component, and a plan for incorporating static analysis and on-line race detection. The integration of this component 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 will improve the coverage quality and the performance of the noise maker. Another example is Java PathExplorer (JPaX) [19]. Java PathExplorer is a runtime monitoring tool for monitoring the execution of Java programs. It automatically instruments the Java bytecode of the program to be monitored, inserting logging instructions. The logging instructions write events revelant for the monitoring to a log file or to a socket in case online-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.

One of the goals of this proposed project is to define a set of APIs so that improvement in one tool could 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 could work on one component, use the rest "off-the shelf" and check the global impact of his work. To be more concrete: assume that an instrumented application is available in which a call is placed in every concurrent location that has information such as the thread name location, bytecode type, abstract type (variable, control), read/write. The writer of a race-detection or noise heuristic can then write his algorithm only.

If the instrumentor is told some information by the static analyzer, on every instrumentation point, this can be used to decide on a subset of the points to be instrumented. For example, only on access to variables touched by more than one thread which exist at this point. Alternatively, the information may be passed through in the instrumented call to the noise maker or any other tool that makes use of the instrumentation.

## 4. Benchmark

The different technologies for concurrent testing may be compared to each other in the number of bugs they can find or the probability of finding bugs, in the percentage of false alarms and in performance overhead. Sometimes the technology itself can not be compared as it is only part of a solution and the improvement in the solution, as a whole, must be evaluated. In order to facilitate the comparison, we propose to create a benchmark that is composed of four parts. The first is a repository of programs on which the technologies can be evaluated, composed of:

- Multi-threaded Java programs including source code and bytecode in standard project format
- Tests for the programs and test drivers
- Documentation of the repository and of the bugs in each program
- Versions of the programs instrumented with calls to empty classes containing information useful to noise, replay, coverage, and race applications.
- Sample traces of executions using the standard format for race detection and replay. Each record in the traces contain information about the location in the program from which it was called, what was instrumented, which variable was touched, thread name, if it is a read or write, and if this location is involved in a bug
- A script for producing any number of desirable traces in the above format. (Possibly the script will have inputs that decide the format as well as the locations at which the script points are written)

The repository of programs should include many small programs that illustrate specific bugs as well as larger programs and some very large programs with bugs from the field. The fact that not only the programs with the bugs are available but also the instrumented program and the traces, makes evaluating many of the technologies much easier. For example, race detection algorithms may be evaluated using the traces without any work on the programs themselves.

The second component of the benchmark is a prepared experiment. One problem in evaluation of technologies like noise makers and race detection is that statistics need to be gathered and analyzed. The question is not if a bug can be found using the technology on a specific test but what is the probability of that bug being found. The experiment part of the benchmark contains prepared scripts with which programs such as race detection and noise can be evaluated as to how frequently they uncover faults, and if they raise false alarms. The analysis of the executions and statistics on the performance of the technologies is also executed with a script. This script produces a prepared evaluation report, which is easy to understand. The bottom line is that we will make it easier to evaluate technology once a technology is ready to be tested. All the machinery will be in place so that with the push of a button, it can be evaluated and compared to alternative approaches.

In the previous section we talked about the technologies with potential for impacting the concurrent testing problem. We showed that the technologies are very interdependent. The third component of the benchmark is a repository of tools with standard interfaces. This way the researcher could use a mix-and-match approach and complement her component with benchmark components to create and evaluate solutions based on the created whole. This repository will include, at the very least, an instrumentor which is needed in most solutions as well as some noise makers and race detection components.

The fourth component is a specially prepared benchmark program that has no inputs and many possible results. We create the program by having a "main" that starts many of our simpler documented sample programs in parallel, each of which writes its result (with a number of possible outcomes) into a variable. The benchmark program outputs these results as well as the order in which the sample programs finished. Tools such as noise makers can be compared as to the distribution of their results. Analysis of outcomes will be produced as part of the prepared experiment.

## 5. Conclusions

In this paper, we discussed the problem of testing multithreaded technology and how to create 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 was made of a variety of relevant technologies. Toward 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 [12] [11] but they do not provide an open interface for extention and do not support the evaluation of competing tools and technologies.

The first phase in preparing such a benchmark is to survey researchers working in the field as to what will be of most use to them. Interesting questions which could be in cluded in the survey are:

- Are there additional technologies that should be included in the benchmark? If so, please describe the technology and how it interacts with the other technologies.

- Would you like additional artifacts to be included in the benchmark?

- Do you have any specific requirements for interfaces of technologies included in the benchmark? Please explain the reason for them.

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 exercize. Indeed, this was one of the motivations of this paper as the work reported in [4] started this way.

A number of groups have expressed interest in participating in this project. We hope to have the specification phase completed by July 2003. We are also looking at formal structures under which this project could be held.

## References


[1] S. Asbury and S. R. Weiner. *Developing Java Enterprise Applications,*. Wiley Computer Publishing, 1999.

[2] G. S. Avrunin, J. C. Corbett, M. B. Dwyer, C. S. Pasareanu, and S. F. Siegel. Comparing finite-state verification techniques for concurrent software. Technical Report UM-CS-1999-069, Department of Computer Science, University of Massachusetts at Amherst, USA, 1999.

[3] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

[4] 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.

[5] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proc. 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 211–230, Nov. 2002.

[6] A. S. Cheer-Sun Yang and L. Pollock. All-du-path coverage for parallel programs. *ACM SigSoft International Symposium on Software Testing and Analysis*, 23(2):153–162, March 1998.

[7] 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.

[8] 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.

[9] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, Welches, Oregon, August 1998.

[10] J. C. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Software Engineering (ICSE)*. ACM Press, June 2000.

[11] J. C. Cunha, P. Kacsuk, and S. C. Winter, editors. *Parallel Program Development For Cluster Computing*. Nova Science Publishers, Jan. 2000.

[12] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002. Also available as `http://www.research.ibm.com/journal/-sj/411/edelstein.html`.

[13] C. Flanagan and S. N. Freund. Detecteing race conditions in large programs. In *Proceedings of the Program Analysis for Software Tools and Engineering Conference*, June 2001.

[14] P. Godefroid. Model checking for programming languages using verisoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[15] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 69–82, 2002.

[16] J. J. Harrow. Runtime checking of multithreaded applications with visual threads. In *SPIN*, pages 331–342, 2000.

[17] A. Hartman, A. Kirshin, and K. Nagin. A test execution environment running abstract tests for distributed software. In *Proceedings of Software Engineering and Applications, SEA 2002*, 2002.

[18] 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.

[19] K. Havelund and G. Rosu. Monitoring java programs with Java PathExplorer. In *In Proceedings First Workshop on Runtime Verification, RV'01, Paris, France, July 23,*.

[20] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 58–70, 2002.

[21] G. J. Holzmann and M. H. Smith. Software model checking: Extracting verification models from source code. In *Proc. International Conference on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV)*, pages 481–497. Kluwer, 1999.

[22] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai. Toward integration of data race detection in dsm systems. *Journal of Parallel and Distributed Computing*, 59(2):180–203, 1999.

[23] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Proc. ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 26–38, 2000.

[24] Y. Malaiya, N. Li, J. Bieman, R. Karcich, and B. Skibbe. Software test coverage and reliability. Technical report, Colorado State University, 1996.

[25] B. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin, Madison, 1995.

[26] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[27] R. Netzer and B. Miller. Detecting data races in parallel program executions. In *Advances in Languages and Compilers for Parallel Computing, 1990 Workshop*, pages 109–129, Irvine, Calif., 1990. Cambridge, Mass.: MIT Press.

[28] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 83–94, 2002.

[29] B. Richards and J. R. Larus. Protocol-based data-race detection. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 40–47. ACM Press, 1998.

[30] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

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

[32] S. D. Stoller. Testing concurrent java programs using randomized scheduling. In *In Proceedings of the Second Workshop on Runtime Verification (RV)*, volume 70(4) of Electronic Notes in Theoretical Computer Science. Elsevier, 2002.

[33] J. A. Whittaker. *How to Break Software*. Addison-Wesley, 2003.