Toward a Benchmark for Multi-Threaded Testing Tools

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

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

Looking for intermittent bugs is a problem that has been getting prominence in testing. 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. We outline a road map for combining the research on the different disciplines of testing multi-threaded programs and on evaluating its quality. The project goals are to create a benchmark that can be used to evaluate different solutions, to create a framework with open API's that enables combining techniques in the multi-threading domain, and 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. The benchmark, apart from containing programs with documented bugs, includes other artifacts, such as traces, that are useful for evaluating some of the technologies. We have started creating such a bench mark and detail the lesson learned in the process. The framework will enable technology developers, for example, race detectors, to concentrate on their components and use other ready made components, (e.g., instrumentor) to create a testing solution.

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 [12]. Having new dual core or hyper-threaded processors in the personal computer, makes the testing of multi-threaded programs even more important. It turns out the programs that use 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 Microsoft has also addressed the issue.

There are a number of distinguishing factors 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. Only a few of the interleavings actually produce concurrent faults. Thus, the probability of producing a concurrent fault is very low. Second, under the simple conditions of unit testing, the scheduler is deterministic; therefore executing the same tests repeatedly does not help. Due to this fact, concurrent bugs are often not found early in the process, rather 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).

All the currently used testing techniques, some of which, such as coverage and inspection, proved very useful, ad-

dress sequential problems. One solution 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 the user 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 the industry. Progress has been made in many domains and it seems that a high quality solution will contain components from many of them. Work on race detection [38] [40] [28] [34] [17] 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 to increase the probability of bugs being discovered by creating more races [43] [15]. The tools that cause races do not report false alarms (they actually do not report anything); they just try to make the user tests fail. Tools for replay [10] are necessary for debugging and contain technology useful for testing. It is hard to create replay that always works. Therefore, tools that increase the probability of replay have also been developed [15]. Static analysis tools of various types, as well as formal analysis tools, are being developed to detect faults in the multi-threaded domain [42] [24] [11]. Analysis tools that show a view of specific interest in the interleaving space both for coverage and performance [7][24] 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 [23] (Rational Robot and Mercury WinRunner). There are a number of approaches to cloning, some of which are very practical. A variety of programming and debugging aids for such an environment are closely related but beyond the scope of this work.

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 [2]. The benchmark we are working on is different in that it not only contains programs against which the tools are evaluated, but 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 if a it is a false negative or a real result. We started working on this benchmark, and already created more then forty annotated programs. This work has taught us how to annotate programs and showed that the programs are useful in uncovering problems in testing tools. The annotation of bugs includes information about location in the program, variables involved, and bug patterns exhibited. The annotation work obviously draws on bug pattern description, however, we also learned about new bug patterns in the annotation work.

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

2. Existing dedicated concurrent testing technologies

In this section we survey technologies that 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 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. The static technologies 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 do 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 primarily 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 in the industry for concurrent bugs. 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 mainly 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 spe-

cial 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.) 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 [27], Bandera [11], SLAM [3], and BLAST [26].

Model checking is infeasible for concurrent systems with very large state spaces. Therefore, the goal is not only to translate the program into the modeling language, but to determine which details of the program are not essential for verifying 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, 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 [8], 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 concurency-related properties. For example, there are specialized type systems [17] [6] [20] and data-flow analyzers [16] [45] for detecting data races; some of these analyses, specifically [6] and [16], also detect deadlocks. The type systems are modular, scalable, and conservative (i.e., they never overlook errors), 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. Static conflict analysis [45] is conservative and automatic, but less scalable than the type systems. RacerX [16] is scalable and automatic but not conservative (it can miss some errors). There are also general verification-oriented static analysis frameworks, such as Canvas [37], ESP [13], and xgcc [21], but they generally do not model concurrency, so they are potentially unsound when applied to concurrent programs. Nevertheless, they may still be effective in practice at finding some concurrency-related errors, e.g., forgetting to release a lock [21]. TVLA [29] is a general static analysis framework that can model concurrency and rigorously verify a variety of properties of concurrent programs, as demonstrated in [48, 49]. 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. 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 level, the bytecode, 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 therefore the most common.

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

Noise makers - A noise maker [15] [43] 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 [32] [47] usually work by denying the application certain 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 perform 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 is 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 [40]. 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 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 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 a 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 exist which can examine traces for evidence of deadlock potentials [22] [25]. 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 to 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 algorithmf, 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 [10] 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 [15], 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 [31] 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). A more concrete example is a coverage model with a task for each synchronization that check that this synchronization has been utilized. A synchronization is utilized if it either stopped another thread or was stopped by it. These two coverage models are implemented in ConTest [15]. Additional coverage measures should be created and their correlation to bug detection studied. A new and interesting research issue uses coverage 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.

Systematic state space exploration - Systematic state space exploration [19] [42] [24] 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 deadlock, 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 [33]. 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 during system testing or as a specialized phase called stress testing. The idea, used in common commercial tools for testing client-server applications such as 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 that require careful design. The first is that the expected results of each clone need to be interpreted, so verifying wheather the test passed or failed is not necessarily straightforward. Many times, changes that distinguish between the clones are necessary. This technique is a purely 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 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 there is even 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

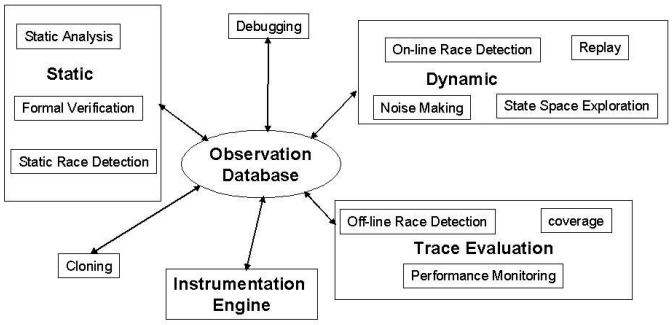

Figure 1. Interrelations between technologies

- Possible race locations - location in the programs that are suspect
- Unimportant locations - areas which are well synchronized, for example only one thread may be alive at that time
- Coverage information - database showing which coverage tasks were covered
- Traces of executions - to be used by off-line analyzers

Instrumentation is the process of automatically modifying code by adding user exits. For example, at every statement increase a counter and you get coverage. Instrumentation is not a multi-threading testing technology but a very important enabling technology. Instrumentation is used by all the dynamic and trace evaluation technologies The instrumentor is told by the observation database what to instrument. 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 [41]. Augmenting AspectJ so it can do all the work required in this framework from an instrumentor is not a very difficult job. As a side remark, the instrumentation technology needed for all the dynamic technologies, and for off-line trace analysis such as off-line race detection, is virtually identical.

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. Choi et al’s work [9, 36] and by von Praun and Gross [45] are nice examples of the use of static analysis to optimize run-time race detection. With the observation database one may improve a race detector while using an existing static analyzer.

The information gleaned in static analysis can also be transfered to the instrumentor and used to reduce the over-

head, (i.e. instrument only what really needs instrumenting), or by attaching information that is used in run-time to the instrumentation calls. For example, Stoller in [43] improved over ConTest by finding 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 a plan for incorporating static analysis and on-line race detection. 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 will improve the coverage quality and the performance of the noise maker. Another example is Java PathExplorer (JPaX) [25], a runtime 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 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.

Creating such technologies can currently be done only in large projects such as ConTest in IBM and JPaX in NASA. 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 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 can 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, without writing the entire system in which it is used.

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, the percentage of false alarms, and 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 proposed creating a benchmark that is composed of two parts:

1. a repository of annotated programs to be described, with which technologies can be evaluated and
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 including:

- Source code (and bytecode) in standard project format
- Test cases, and test drivers
- Documentation of the bugs in each program
- Instrumented versions of the programs to be used by noise, replay, coverage, and race applications
- Sample traces of program executions. Each record in the traces contains information about the location in the program from which it was called, what was instrumented, which variable was touched, thread name, and whether it is a read or write.

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.

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 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, at the very least, an instrumentor, which is needed in most solutions, as well as some 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 to novice programmers, this was just a beginning.

As part of the course, the students studied the taxonomy of concurrent bug patterns [35] and technologies for finding such bugs [39, 14, 43, 42, 44]. 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 offered relevant practical experience. All the bugs in the program were to be documented and points would be taken off for every undocumented bug found. They were also asked to write a report describing the program and its functions, bugs, and possible outcomes. The assignment can be seen in (http://cs.haifa.ac.il/courses/softtest/testing2003/).

Most of the students chose to write non-atomic bug patterns, mostly 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. Few of the students created bugs that were dependent on interleavings that happen only with specific inputs.

Using raceFinder [5] to produce test reports, and working with a large number of different users, allowed us to study this tool further, 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 several important lessons:

- 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, supports 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 [48, 8], dynamic analysis [18, 46], or both [9] to reduce the number of program locations. This idea is now being implemented in raceFinder with good results.

- 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-threaded 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).

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 [10, 14].

- The input of the program (control flow) - some of the interleavings that induce a concurrent bug are input-dependent.

- The design of the program contains a fault (not the implementation) and this fault manifests itself 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. 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, 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 [30]), and it may be interesting to combine a noisemaker 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 student non-standard programming practices, which were not considered in the tool design. Bugs included:

- Some bugs in the implementation (mainly in the GUI for entering noise parameters) 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.
- When running the program instrumented with raceFinder, even slight changes in the program timing can cause the program to run endlessly.
- RaceFinder currently doesn't offer enough debugging information to understand which interleaving caused a bug to manifest and why.
- An interesting bug was found using ConTest. ConTest has a heuristic called halt-one-thread. When this heuristic 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 at this point for a very long time. 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, since when 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 thread 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.

Our benchmark is off to a good start with almost 40 programs, about half of which were created by students and the rest by tool makers and taken of the Internet, mainly open source. We would like to extend the benchmark with additional student programs. Hopefully, other universities will join the effort.

5. Conclusions

In this paper, we discussed the problem of evaluating multi-threaded testing 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 [15] [12] 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 [4] started this way.

We discussed this project in PADTAD 2003 and PADTAD 2004 and with additional groups such as 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 had 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 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 toolmaker, 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 experienced 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 we will have the bug and the correct fix for the benchmark, which will be useful for checking 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. 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 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. 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 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.

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] Y. Ben-Asser, Y. Eytani, and E. Farchi. Heuristics for finding concurrent bugs. In *In the International Parallel and Distributed Processing Symposium, IPDPS 2003, PADTAD workshop*, April 2003.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verification in polynomial time. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 57–68. ACM Press, 2002.

[14] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Testing multi-threaded java programs. *IBM System Journal Special Issue on Software Testing*, 41(1), February 2002.

[15] 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`.

[16] D. R. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proc. 19th ACM Symposium on Operating System Principles (SOSP)*, pages 237–252. ACM Press, Oct. 2003.

[17] 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.

[18] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *31'st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 2004.

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

[20] D. Grossman. Type-safe multithreading in Cyclone. In *Proc. ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 13–25. ACM Press, 2003.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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,*.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] B. Long, D. Hoffman, and P. Strooper. tool support for testing concurrent java components. *IEEE Transactions on Software Engineering*, 29(6), June 2003.

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

[32] 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.

[33] 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.

[34] 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.

[35] Y. Nir, E. Farchi, and S. Ur. Concurrent bug patterns and how to test them. In *In the International Parallel and Distributed Processing Symposium, IPDPS 2003, PADTAD workshop*, April 2003.

[36] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proc. ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2003.

[37] 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.

[38] 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.

[39] S. Savage. Eraser: A dynamic race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

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

[41] A. Schmidmeier, S. Hanenberg, and R. Unland. Implementing known concepts in aspectj.

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

[43] 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.

[44] G. Vijayaraghavan and C. Kaner. Bug taxonomies: Use them to generate better tests. In *Star East 2003, Orlando, Florida*, May 2003.

[45] C. von Praun and T. Gross. Static conflict analysis for multithreaded object-oriented programs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 115–128. ACM Press, 2003.

[46] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *Proc. Third Workshop on Runtime Verification (RV)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2003.

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

[48] E. Yahav. Verifying safety properties of concurrent java programs using 3-valued logic. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*, pages 27–40. ACM Press, 2001.

[49] E. Yahav and M. Sagiv. Automatically verifying concurrent queue algorithms. In *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.