

Guided test for detecting concurrency errors

Neha Rungta
Department of Computer Science
Brigham Young University
Provo, UT 84601
neha@cs.byu.edu

Eric G. Mercer
Department of Computer Science
Brigham Young University
Provo, UT 84601
eric.mercer@byu.edu

1. PROBLEM DESCRIPTION

The ubiquity of multi-core Intel and AMD processors is prompting a shift in the programming paradigm from inherently sequential programs to concurrent programs to better utilize the computation power of the processors. The shift from sequential programs to concurrent programs is accompanied with significant challenges. Although parallel programming is well studied in academia, research, and a few specialized problem domains, it is not a paradigm commonly known in mainstream programming. As a result, there are few, if any, tools available to programmers to help them test and analyze concurrent programs for correctness.

The state of the art testing and verification techniques are insufficient for testing and verifying the presence of concurrency errors in multi-threaded programs. The current techniques lie on various points on a spectrum of precision and scalability. On one end of the spectrum, static analysis techniques [18, 8, 6, 20, 11, 1] are imprecise but can scale to large systems. The analysis techniques mostly ignore the semantics of the program and analyze the source code to report warnings about possible errors. The errors reported by static analysis techniques have to be manually verified which is difficult, tedious, and sometimes not possible. At the other end of the spectrum, model checking techniques, [10, 19, 2], are precise and sound but do not scale to large systems. Model checking techniques exhaustively enumerate all possible behaviors of the system to verify the correctness of the system. The exhaustive nature of model checking leads to a huge state space explosion that makes model checking intractable for verifying most applications.

We propose a guided test technique to automatically verify deadlocks, race conditions, and other safety violations like exceptions and assertion violations from warnings generated by imprecise static analysis techniques. The intuition is to use the imprecise analysis techniques to create an initial set of candidate error traces, and then use a precise technique, guided test, to verify the error traces and produce concrete

program executions manifesting the real errors. Our proposed solution would thus present to the developer a fully verified set of real deadlocks, race conditions, and safety violations with concrete error traces. Although the error traces generated by the guided test technique are real and require no further verification; if it does not find an error we cannot prove the absence of the error; thus, our proposed guided test technique is sound in error detection but not complete. In essence, the proposed guided test technique attempts to bridge the gap between precision and scalability while detecting concurrency errors in multi-threaded programs.

Thesis Statement: A guided test technique that directs the program execution through a sequence of locations, generated using output from static analysis tools, while controlling thread-schedules and data input values can verify the feasibility of warnings generated by the static analysis tools.

Overall Test Technique: We present the overall guided test technique in Figure 1. The input to the guided test technique is the program and the set of locations. During a greedy depth-first search, we first prune the set of possible thread-schedules, rank the thread-schedules, and generate the data values to guide the program execution through the sequence of locations. At the end of the test, there are two possible answers: (1) Yes—Error Trace, (2) Do not know. Within the specified constraints of time and memory, if the test finds an error, the counterexample is reported specifying the conditions which lead to the error; otherwise, the guided test cannot conclusively determine the feasibility of the warning.

2. APPROACH

We have designed a prototype implementation of the guided test technique to validate its effectiveness in detecting concurrency errors in multi-threaded programs [17]. In the prototype implementation, a guided test directs the program execution through the sequence of program locations (partial error trace) manually generated from warnings reported by static analysis tools such as FindBugs [11] and Jlint [1]. The sequence may or may not be feasible in any valid execution of the program. The sequence, however, serves as a starting point for verifying a possible error.

We use the Java Pathfinder (JPF), [19], model checker as a test execution engine. JPF allows us to systematically explore and rank the behaviors of the program relevant to the error being tested. JPF represents the behaviors of the sys-

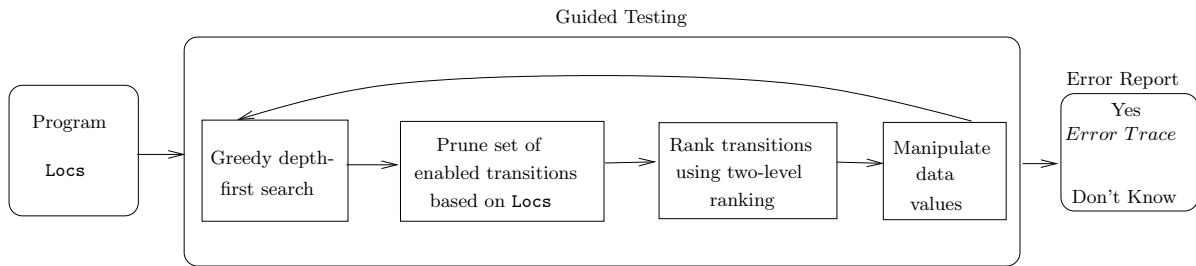


Figure 1: The proposed guided testing approach

tem as a set of states and transitions. A state in the program is the snapshot of the program condition which consists of the values of variables at a specific program location and the heap; while the transitions represent the change in the program condition from one set of variable values (including the heap) to another as a result of the executing byte-codes.

The guided test directs the program execution in a greedy depth-first manner using a two-level ranking scheme for choosing successors. The two-level ranking is based on the following criteria: (1) the number of locations in the partial error trace already observed; and (2) the perceived cost to reach the next location in the error trace. Successor states that have observed a higher number of locations from the sequence are ranked as more *interesting* compared to states that have observed fewer locations in the sequence. A secondary heuristic estimate based on the proximity to the next location in the sequence is used to rank the successor states that have observed the same number of locations from the sequence. The guided test uses the two-level ranking to pick the best successor of a given state in a depth-first search.

Distance estimate heuristics [12, 13] lend themselves naturally to guide the test toward the next location in the program. Distance estimate heuristics compute the number of transitions between the current location and the next location in the sequence using the control flow representation of the program with partial context information. State of the art distance heuristics [5, 12, 13] cannot accurately compute the distance estimates in the presence of polymorphism. In this work we design a distance estimate heuristic to compute the distance estimates in polymorphic programs [16]. The technique to compute the distance heuristic uses a rapid type analysis to reduce the number of unresolved polymorphic method types in the program. It then performs an interprocedural static analysis to conservatively compute distance estimates. Finally, to compute the heuristic value it dynamically refines the distance estimates when the types of polymorphic methods are resolved at transaction boundaries during program execution. Note that transaction boundaries are program locations where there are non-deterministic choices due to thread schedules and data values.

3. RESULTS

In Table 1 we present some initial results that demonstrate the effectiveness of our prototype implementation of the guided test technique [17]. The `AbsList`, `Deadlock`, and `AryList` examples in Table 1 are programs that use the JDK

Subject	Random DFS	Guided Test		
		PFSM	Const	FSM
TwoStage(7,1)	0.41	1.00	1.00	1.00
TwoStage(8,1)	0.04	1.00	1.00	1.00
TwoStage(10,1)	0.00	1.00	1.00	1.00
Reorder(9,1)	0.06	1.00	1.00	1.00
Reorder(10,1)	0.00	1.00	1.00	1.00
Wronglock(1,20)	0.28	1.00	1.00	1.00
AbsList(1,7)	0.01	1.00	0.37	-
AbsList(1,8)	0.00	1.00	0.08	-
Deadlock(1,9)	0.00	1.00	1.00	-
Deadlock(1,10)	0.00	1.00	1.00	-
AryList(1,5)	0.81	1.00	1.00	-
AryList(1,8)	0.00	1.00	1.00	-
AryList(1,9)	0.00	1.00	1.00	-
AryList(1,10)	0.00	1.00	1.00	-

Table 1: Defect Detection Rate.

1.4 library in accordance with the documentation. The particular usage, however, manifests concurrency errors in the JDK 1.4 library. We use Jlint [1] on these models to generate warnings on possible concurrency errors in the JDK 1.4 library. At present we manually generate a short sequence of locations that are relevant to verifying the feasibility of the warning. In Table 1 we report the error density of the model given a particular technique. The error density is defined as the probability of a technique finding an error in the program. To compute this probability we use the ratio of the number of error discovering trials over the total number of trials executed for a given model and technique. For the models presented in Table 1, we execute a 100 trails where every trial is time bounded at one hour. A technique that generates an error density of 1.00 is termed as effective in error discovery while a technique that generates an error density of 0.00 is termed as ineffective. Note that we run multiple trails of the guided test because all ties in heuristic values are randomly broken. In an extensive study, [14], we show that randomly breaking ties in heuristic values results in better error discovery rates and effectively controls for default search order.

The first column (`Subject`) in Table 1 specifies the name of the subject and the thread configuration used. The thread configuration is the number of threads for each type of thread in the model. The second column labeled `Random DFS` shows the error densities generated by randomized depth-first search.

The last three columns under **Guided Test** are the error densities generated by the guided test technique using a fixed sequence length and three different secondary heuristics: PFSM heuristic (**PFSM**), constant heuristic (**Const**), and FSM heuristic (**FSM**). The PFSM heuristic is the new distance heuristic designed to compute distance estimates in the presence of unresolved polymorphic methods [16]. For the entries indicated by “-” in the FSM heuristic column, the static analysis does not finish within the time bound of one hour. The results in Table 1 indicate that guided test, overall, has a better defect detection rate when compared to randomized depth-first search. In the different models as we manipulate the number of threads to increase the semantic hardness of discovering an error in the model (as defined in [15]), the randomized depth-first search struggles to find the error while the guided test technique using the PFSM heuristic finds an error in every single trial as indicated by the error density of 1.00. Even the constant heuristic outperforms the randomized DFS in most models. A similar trend is observed in the other measures, like states generated, total time taken, and total memory used in the error discovery trials. Detailed comparisons are presented in [17, 16].

The initial results presented in Table 1 suggest that guided test is successful in quickly verifying errors in the JDK 1.4 concurrent library using sequences generated from static analysis warnings and has the potential to scale to practical concurrent applications.

4. PROPOSED RESEARCH

In this section we present an overview of techniques that can be used to improve and refine the existing guided test prototype implementation. One, we need a technique to generate the sequence of locations or a partial error trace automatically from the output of static analysis tool. Two, we need to aggressively prune the number of thread-schedules. And, three, there is a need for a technique that handles data non-determinism and restricts the possible data input values.

Generating a sequence of locations: We propose to design and implement a pre-processing phase that uses the output of static analysis tools to automatically generate a sequence of locations that are relevant to verifying the feasibility of the warning. We use an example to demonstrate how the output of a static analysis tool can be used to generate a sequence of locations. The static analysis tools use the lockset analysis, [6, 20], to detect potential deadlocks in multi-threaded programs caused by cyclic lock dependencies. The lockset analysis generates constraints based on the sequence of locks acquired in the program; if, one thread acquires lock a and then lock b in the control flow representation of the program then the lockset analysis generates a constraint $a \rightarrow b$. In some other part of the program if the lockset analysis generates the constraint $b \rightarrow a$, then the two constraints represent a potential deadlock due to a cyclic dependency. Recall that the lockset analysis does not reason whether the control paths along the cyclic lock dependencies are on feasible execution paths for the program. We can generate a sequence of locations from the start of the program to the location where lock a is acquired in the first constraint, and also a sequence of locations from the start of the program to where lock b is acquired in the second

constraint. Similarly, we can design techniques to automatically generate the relevant sequence of locations for warnings other than deadlocks that are reported by the static analysis tool.

Pruning possible thread-schedules The current guided test technique uses a dynamic partial order reduction. Partial order reduction techniques, [7], reduce the number of thread-schedules that need to be explored to show correctness of the program. Current partial order reduction techniques consider all possible dependencies between shared variables in multi-threaded programs. In this work, we propose a modular partial-order reduction technique that aggressively prunes the interleavings arising from dependencies that are not related to guiding the program execution toward the next location in the sequence.

Generating Data values: To handle data non-determinism, we propose to build a set of constraints while guiding the execution of the program. At data non-determinism points, the guided test assigns random values to input variables, while at branch points, the test generates constraints to incrementally build a propositional satisfiability formula. The guided test, at branch points, also checks which branch of the conditional statement has a better heuristic value. If the current set of random input data values allows the execution to go down the branch with the better heuristic value then we continue guiding the program; otherwise, we generate a new value for the input variable that allows us to execute the branch statement with the better heuristic value.

Further Evaluation: We propose to evaluate our proposed technique against other dynamic test techniques such as ConTest and RaceFinder. ConTest and RaceFinder are dynamic analysis techniques that use various heuristics to detect concurrency errors in multi-threaded programs. Like the guided test technique, ConTest and RaceFinder are sound in error discovery but are not complete. We hope to achieve better error discovery rates when compared to ConTest and RaceFinder. We also hope to find errors that cannot be discovered with traditional test techniques.

Quantifying Test Effort: In the case when the guided test is unable to detect the error in the program, we propose to rank the static analysis warning using a probabilistic measure. The measure will be used to indicate the likelihood of the error actually existing in the program based on the data gathered during the guided test trials. In other words we will quantify the test effort in the absence of error discovery. Note that other test techniques such as ConTest and RaceFinder report statement and branch coverage; however, we will define a coverage measure based on the probability of the static analysis warnings being feasible.

5. RELATED WORK

Static analysis techniques mostly ignore the semantics of the program and reason about errors by simply analyzing the source code of the program. RacerX, [6], does a top-down inter-procedural analysis starting from the root of the program. Similarly, the work by Williams *et al.*, [20], does a static deadlock detection for Java libraries. FindBugs [11] and JLint [1], looks for suspicious patterns in Java programs. The warnings reported by static analysis techniques have to

be manually verified. Such an exercise becomes intractable for large programs. The output, however, serves as ideal input for the guided test technique proposed in this paper.

The deterministic execution technique, [9], used to test concurrent Java monitors is related to the guided test technique. The deterministic execution approach, however, requires a significant manual effort. The tester is required to manually specify the data values that execute the different branch conditions, the possible interactions between the threads, and the sequence of methods. The program is executed using the manually generated inputs to check whether it leads to an error. In contrast, our work proposes to automatically generate the sequence of methods using the output of static analysis tools; also we propose to intelligently rank thread schedules and pick data values at non-determinism points to force the execution along the sequence of methods to test feasibility.

There have been various techniques that use the output of static analysis tools for automatic test case generation. Check ‘n’ Crash, [3], is a hybrid test technique that uses a constraint solver to generate concrete test cases based on the output from the static analyzer tool—ESC/Java. DSD crasher [4], extends Check ‘n’ Crash by adding information from a runtime analysis tool. Check ‘n’ Crash and DSD crasher, however, only generate test cases for safety violations in sequential programs. We take the same idea and extend it for concurrent programs in that we take the output of static analysis tools and verify the feasibility of the warnings using a more precise guided test technique.

6. REFERENCES

- [1] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded java programs. In *ASWEC '01: Proceedings of the 13th Australian Conference on Software Engineering*, page 68, Washington, DC, USA, 2001. IEEE Computer Society.
- [2] T. Ball and S. Rajamani. The SLAM toolkit. In G. Berry, H. Comon, and A. Finkel, editors, *13th Annual Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Paris, France, July 2001. Springer-Verlag.
- [3] C. Csallner and Y. Smaragdakis. Check ‘n’ Crash: combining static checking and testing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 422–431, New York, NY, USA, 2005. ACM Press.
- [4] C. Csallner and Y. Smaragdakis. DSD-Crasher: a hybrid analysis tool for bug finding. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 245–254, New York, NY, USA, 2006. ACM.
- [5] S. Edelkamp and T. Mehler. Byte code distance heuristics and trail direction for model checking Java programs. In *Workshop on Model Checking and Artificial Intelligence (MoChArt)*, pages 69–76, 2003.
- [6] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, New York, NY, USA, 2003. ACM Press.
- [7] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 110–121, New York, NY, USA, 2005. ACM.
- [8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.
- [9] C. Harvey and P. Strooper. Testing Java monitors through deterministic execution. In *ASWEC '01: Proceedings of the 13th Australian Conference on Software Engineering*, page 61, Washington, DC, USA, 2001. IEEE Computer Society.
- [10] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
- [11] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.
- [12] N. Rungta and E. G. Mercer. A context-sensitive structural heuristic for guided search model checking. In *20th IEEE/ACM International Conference on Automated Software Engineering*, pages 410–413, Long Beach, California, USA, November 2005.
- [13] N. Rungta and E. G. Mercer. An improved distance heuristic function for directed software model checking. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 60–67, Washington, DC, USA, 2006. IEEE Computer Society.
- [14] N. Rungta and E. G. Mercer. Generating counter-examples through randomized guided search. In *Proceedings of the 14th International SPIN Workshop on Model Checking of Software*, pages 39–57, Berlin, Germany, July 2007. Springer-Verlag.
- [15] N. Rungta and E. G. Mercer. Hardness for explicit state software model checking benchmarks. In *5th IEEE International Conference on Software Engineering and Formal Methods*, pages 247–256, London, U.K, September 2007.
- [16] N. Rungta and E. G. Mercer. A distance heuristic for polymorphic programs, 2008. Technical Report, available online at <http://vv.cs.byu.edu/publications/papers/polymorphic.pdf>.
- [17] N. Rungta and E. G. Mercer. Guided test for detecting concurrency errors, 2008. Technical Report, available online at <http://vv.cs.byu.edu/publications/papers/guided-test.pdf>.
- [18] N. Sterling. Warlock— a static data race analysis tool. In *USENIX Technical Conference Proceedings*, pages 97–106, 1993.
- [19] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *15th International Conference on Automated Software Engineering (ASE 2000)*, Grenoble, France, September 2000.
- [20] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 602–629, Glasgow, Scotland, July 27–29, 2005.