Compiling a Benchmark of Documented Multi-Threaded Bugs

Yaniv Eytani
Shmuel Ur

IBM Labs in Haifa, Software and Verification Technology

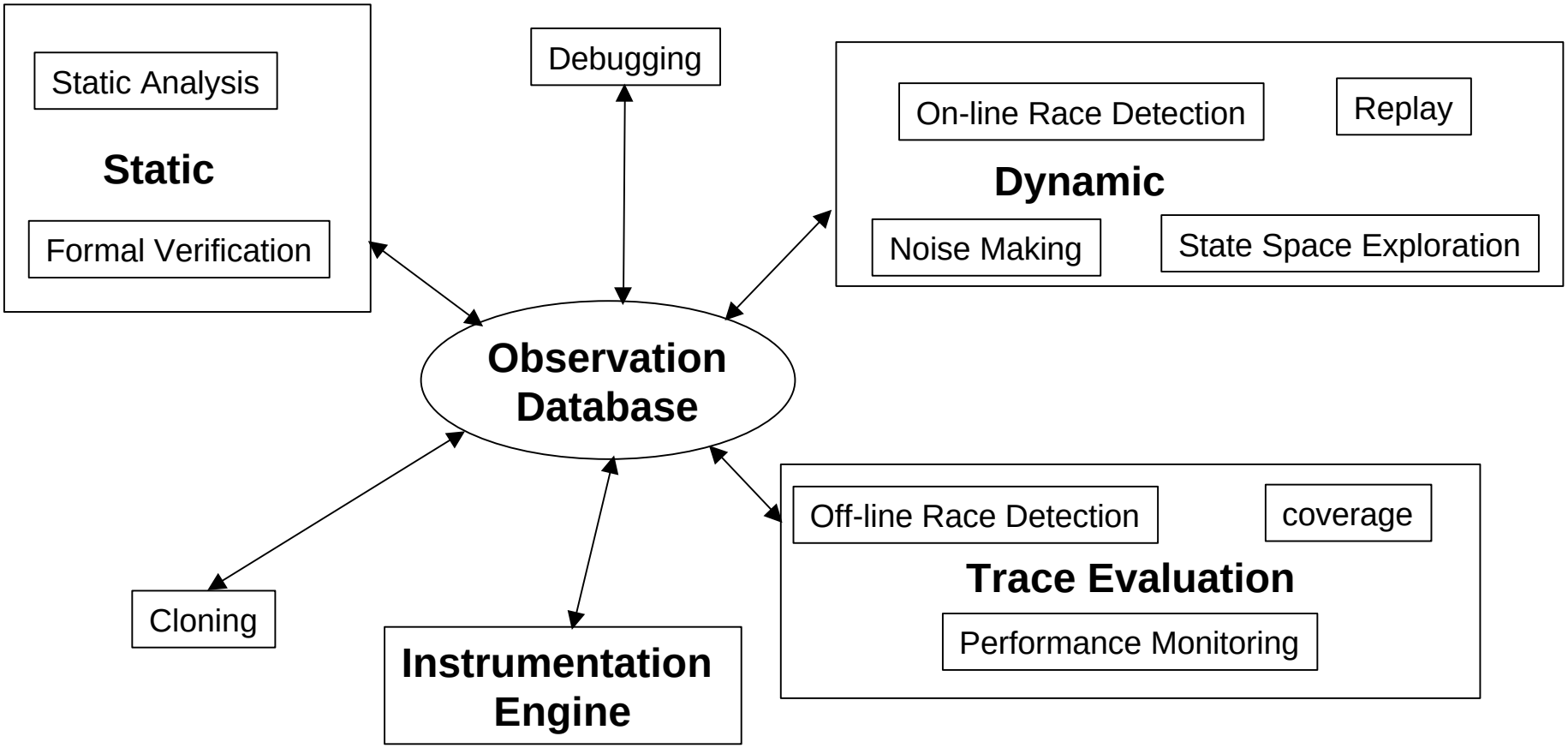
Agenda

- Multi-threaded testing is different
- Background – Last year proposal
- The assignment and what the student wrote
- Lessoned learned
- Conclusions

Distinguishing Factors for Concurrent Testing

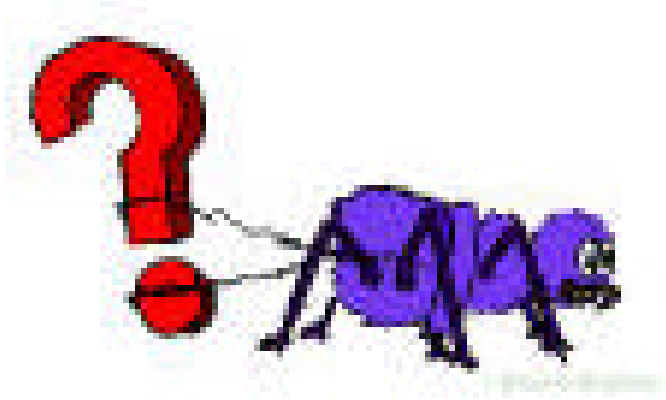
- The set of the possible interleaving is huge
- The bugs are intermittent (now you see me, now you don't)
- Usually the schedulers are deterministic
  - Running the test many times will not help
- Bugs are very hard to fix
  - Hard to recreate
  - Debugging changes the timing
- Current testing technologies are geared to sequential code
  - Inspection
  - Coverage
- Result – Bugs are found very late or by the user

Proposal part I – Framework for interacting technologies



Proposal Part II – Buggy Programs for testing the technologies

- Sample programs to show different bug categories
- Real programs in which bugs were found
- For each program
  - Annotation of the bug
    - Where it is
    - What variables are involved
    - Classification of the bug
  - Expected results, correct, failures, what the failures indicate
  - Instrumented program (generic instrumentation)
- Driver for the test suite and result analyzer



The Assignment – High level Overview

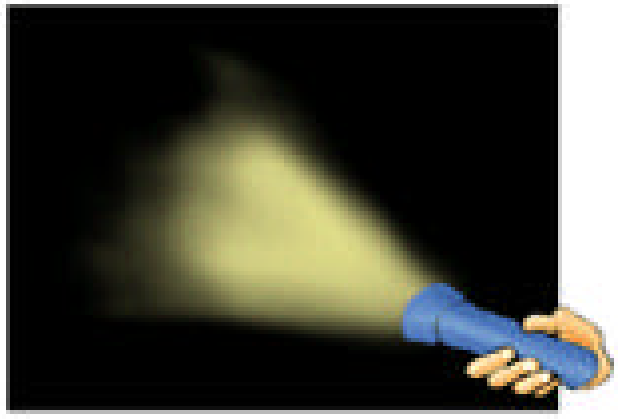
- The course included presentations about bug patterns in multi-threaded programs and bug taxonomies
  - Bug pattern example - lost notify - when the notify is done before the wait and therefore is not used
- We assigned students in a software testing class to write buggy multi-threaded Java programs and document the bugs
  - Each program sketch was approved prior to implementation

The Assignment - Technical Description

- The program should perform some useful task
- The program should include multi-threaded intermittent bugs
- The program should have two parameters as inputs:
  - The name of the output file.
  - Parameter of concurrency {little, average, lot}
- Have the program output tuples of the form into the output file <program name, result, bug pattern indicated>
- A README file explaining the program and its function; the bug and its location; the bug pattern; the relevant variables
- A dictionary file containing all the possible output tuples

Noise Makers

- Cause different interleavings by adding random delays
- Try to make the application behave in an unpredictable way
- Do not report to the user – no false warnings
- Benefit from multiple execution of tests
- Do not guess what the correct result is



The Assignment - Extra credit for testing with raceFinder

- The students who used raceFinder were asked to
  - Try the different features of raceFinder,
  - Evaluate its effectiveness in manifesting a concurrent bug of the pattern in the student's program
  - Manifesting the bug manually (optional)
  - Write a research report
    - Explain their findings.
    - Show the background of the specific bug pattern
    - Show the interleaving that manifested the bug
    - Show statistics evaluating raceFinder features efficiency

Program the students wrote

- The students wrote mostly short programs (100 – 600 lines of code), and most of that code was related to the bug
- The input for most of the programs was created randomly, so the students did not pay too much attention to the results section of the output
- Most of the students didn’t submit clear running instructions for the program
- Clearly, the programs created by the students are biased toward bugs typical to novice programmers

Bugs created for the benchmark

- Most of the students chose to write non-atomic bug patterns (15)
- Other bug patterns used include:
  - Deadlock (3)
  - Sleep bug pattern (2)
  - Orphaned thread bug pattern (2)
  - Double checked locking (1)
  - Notify instead of notifyAll (1)
  - Blocking critical section (3)
  - Bugs that were dependent on interleavings that happen only with specific inputs (2)

Unintentional bugs inserted by the students

- Several multi-threaded bugs (found by ConTest)
- Many of the students did not write the program from scratch, However, all of them had to write a wrapping that would make the program behave according to the specification.
  - Most did not check that the inputs conformed to specification.
  - Many did not write the outputs correctly.
- None decided to document them rather then fix unintentional bugs
  - Lacking experience the students thought that fixing is easier…
  - Alternatively they may not have wanted others to see their mistakes

Insight gained

- Based on the raceFinder reports, we find that creating noise in a few places is more effective for finding concurrent bugs, than creating noise in many program locations.
- raceFinder can effectively handle non-atomic bugs
- Writing multi-threaded code is very hard—even the small programs contained undocumented bugs and buggy interleaving
  - It was sometimes hard for us, despite our previous experience, to understand the undocumented buggy interleavings

Noise Making is not a Silver Bullet

- Bugs that cannot be uncovered with ConTest or raceFinder.
  - For example, if a bytecode is not atomic, the tools cannot create noise inside that bytecode.
  - Bugs related to two-tier memory hierarchy, will not be observed on a one-tier memory Java implementation, regardless of the scheduling
  - Bugs that have to cope with scheduling inside the JVM- for example, the order in which the waiting threads are awakened by a notify

Problems Discovered in raceFinder

- Some bugs in the implementation (mainly the input to noise interface) of raceFinder were discovered and fixed
- When there are many memory accesses in the program, delays (e.g., wait) caused overhead, which made it impossible to test the program.
  - Need to reduce the number of noised memory location
- When running the program instrumented with raceFinder, even slight changes in the program timing can cause the program to trash and run endlessly
- RaceFinder currently doesn't offer sufficient debugging information to give a good understanding of which interleaving caused a bug to manifest and why. Bug related noised locations are not always enough

Problem that was Discovered in ConTest

- ConTest has a heuristic called halt-one-thread
  - 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
- halt-one-thread allows us to stay at a point until the rest of the program cannot proceed - useful for creating unlikely timing scenarios.
  - We reasoned that this couldn't create a deadlock - as soon as there is no progress, the halted thread continues
- Students showed us we were wrong ☹

Future work on expanding the benchmark

- The students should write larger programs with code that perform tasks not related directly to the concurrent bug to allow testing the scalability of the testing tools
- The benchmark should be balanced, having at least a few programs from each bug pattern
- The students should get more education on multi-threading issues
- The students should use more than one testing tool (and tool type), and explore the integration of these techniques
- Giving the students a standard running environment for testing their programs should be considered

Defining the benchmark standards

- The input and the output interface. In addition to documentation, we should provide APIs in order to reduce customization work
- The structure of the submitted report
  - The report should include the program execution instructions and comprehensive description about its structure. An example and a template should be made available
- Creation a standard for a trace
  - Adding traces of programs to the benchmark

Conclusions about the Students

- Our undergraduate software testing class students were given an assignment to write a potentially useful program containing one (or more) concurrent bugs
  - They were told that additional bugs are permissible, but that all the bugs in the program should be documented.
  - They were also asked to write a report describing the program and its functions, bugs, and possible outcomes.
- The assignment turned out to be more difficult then expected. The open nature of the assignment confused quite a number of students
  - The alternative was to do 80 hours of testing work on a project
- None documented the bugs they created by mistake…
- Our conclusion is that even though we stressed the difference between testing and development for the entire course we did not drive it home

Conclusions about testing tools

- In testing the homework assignments we found some bugs in our tools
  - Mainly because the students programmed in ways we never even considered
- 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
- We saw a number of bugs that cannot be uncovered with tools such as ConTest or raceFinder
  - Some of these bugs may actually be masked by the tools

Conclusions about creating a benchmark

- We now know how to define the students' assignment more clearly, and where the pitfalls are expected to be.
- We also have a better idea how to further expand the benchmark
- We expect to get additional feedback from benchmark users which we will use in the next iterations of this exercise