IMUnit: Improved Multithreaded Unit Testing

Position Statement

Vilas Jagannath, Milos Gligoric, Dongyun Jin, Grigore Rosu, Darko Marinov
Department of Computer Science, University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{vbangal2, gliga, djin3, grosu, marinov}@illinois.edu

ABSTRACT

This position paper argues for an approach to bring several techniques successful for (regression) testing of sequential code over to multithreaded code. Multithreaded code is getting increasingly important but remains extremely hard to develop and test. Most recent research on testing multithreaded code focuses solely on finding bugs in one given version of code. While there are many promising results, the tools are fairly slow (as they, conceptually, explore a large number of schedules) and do not exploit the fact that code evolves over several versions during development and maintenance.

Our proposal is to allow explicit specification of *relevant schedules* (either manually written or automatically generated) for multithreaded tests, which can substantially speed up testing, especially for evolving code. To enable the use of schedules, we propose to design a novel language for specifying schedules in multithreaded tests, and to develop tools for automatic generation of multithreaded tests and for improved regression testing with multithreaded tests.

1. PROBLEM AND MOTIVATION

With the advent of the multi-core computing era, parallel programs are becoming the norm rather than the exception. The currently dominant paradigm for developing parallel programs is that of multithreaded code with shared memory, which is prone to data races, atomicity violations, deadlocks, and related bugs. Improved unit testing of multithreaded code has the potential to significantly decrease the number of these bugs and increase the quality of code.

Software testing is the most widely used approach for detecting bugs in practice. We propose an approach that could substantially improve testing of multithreaded code by leveraging both on proven successes in testing of sequential code and on recent advances in checking multithreaded code. For testing sequential code, researchers and practitioners have developed a number of approaches and tools that developers have adopted to help in producing more reliable code. Some of the most successful techniques and tools include unit-testing frameworks such as JUnit for Java (which automates execution of tests and reporting of results, especially for regression testing which reruns tests after a code change), regression test selection [6,12] (which determines what tests to rerun after changing code), and test prioritization [10,11] (which determines in what order to run the tests to find bugs faster). The existing approaches work fairly well for sequential code, but unfortunately do not work nearly as well if translated directly for multithreaded code.

For checking multithreaded code, there has been recently a lot of promising research. Some of the testing tools include CalFuzzer, CHESS, ConTest, and CTrigger. (See [5] for some references.) Since multithreaded code can have different behaviors for different thread schedules, these tools conceptually explore a large number of schedules. As a result, the tools typically take fairly long to check code. Moreover, the existing tools check one given version of code at a time, and do not exploit the fact that code evolves over several versions during development and maintenance. Contrast the focus on checking one version for multithreaded code with regression testing (which validates evolving code) for sequential code.

In other words, while these testing tools are helpful for detecting and replaying bugs, they are suboptimal for ensuring that these bugs are fixed and do not reappear in future versions of the program. The reasons for this are twofold. First, the exploration of a test for multiple possible schedules is much more costly than the normal single execution of a test. Therefore, these tools cannot be used frequently as test runners in a continuous testing or regression testing setup. Second, these tools do not capture the detected bugs in the form of tests that can be quickly re-executed against future versions of the program. For example, a tool could record a schedule that leads to some bug, but a patch for that bug may, say, add a lock which would invalidate the schedule. Therefore, when a program is modified, the tests have to be completely re-explored to detect any regressions.

2. PROPOSED SOLUTION

To improve testing of multithreaded code in face of program evolution, we argue that it is necessary to develop a set of new techniques and tools for multithreaded tests. A multithreaded test is a piece of code that creates and executes two or more threads (and/or invokes code under test that itself creates and executes two or more threads). Executing a test follows some schedule for the execution of the multiple threads (and different schedules can give different

IWMSE '10, May 1 2010, Cape Town, South Africa

results). The key of our approach is to allow *specifying a set of relevant schedules* for each test. These schedules can be written manually by the developer/tester or generated automatically by tools. Our approach conceptually generalizes traditional testing, which can be obtained as an instance by specifying *all possible* (semantically valid) schedules.

Exploring a set of schedules can be costly, based on the size of the set. Our approach allows specifying a set for each test, thus controlling the trade-off between exploration time and thoroughness, effectively controlling the cost and expected benefit. In the limit on one side—(implicitly) specifying all possible schedules—the exploration may be prohibitively expensive (if not infeasible) but provides the most thorough check. In the limit on the other side—specifying a singleton set—the exploration of exactly one schedule lacks coverage but is actually very important for debugging (to replay/reproduce a particular schedule).

We consider three important challenges in specifying and exploiting schedules for multithreaded tests:

(1) Test schedules: *How to specify a set of schedules?* We propose to develop a novel language that makes it easy to specify schedules. The basic entity is an *event* that an execution can raise at various points (e.g., a thread getting blocked or a shared variable being accessed). A schedule is then a (monitorable) property on the sequence of events (e.g., a desired or undesired order of events). The language also allows composition of sets (i.e., building larger sets from smaller sets). *How to execute the code for the given set of schedules?* If a specification is imperative (i.e., telling how to proceed from one thread to another), code can be simply run as is; if a specification is declarative (i.e., only stating the properties of event sequences), the exploration requires some search to find an appropriate schedule. A question related to specifying and executing a set of schedules is what schedules from the set to explore. We leave it up to the user to instruct the tool how much to explore: it can be one/any schedule from the set or some/few specific ones or several randomly selected ones or even all schedules when it is feasible to explore the entire set. We plan to use JavaMOP [2] for monitoring and controlling code executions.

(2) Test Generation: *How to automatically generate schedules for a given testing code?* While we expect developers/testers to manually write high-level schedules to guide tools in exploration, it would be impractical to manually write low-level schedules that contain too many events, e.g., the sequence of all thread context switches in a test execution. We plan to build on previous work on jPredictor [3], a hybrid dynamic and static tool aimed at finding concurrency bugs. While jPredictor found many bugs in real programs, it currently does not generate sets of schedules that lead to bugs. *How to automatically generate both the testing code and the schedules?* Experience with jPredictor shows that multithreaded tests often involve a lot of code and have long executions, but the bugs found can be located in small pieces of code. For example, a test may exercise several methods from some class, whereas the bug occurs because of mis-synchronization between only two methods. We will develop techniques and tools that build smaller tests from bigger ones, as an aid in debugging multithreaded code.

(3) Regression Testing: *How to perform test selection for multithreaded tests (with schedules)?* After developers change the code under test, they typically run the tests to validate that there are no regression errors. Regression selection techniques [6,12] make this process faster by selecting only a subset of tests to run. This approach is successfully used in practice [11], but the existing selection techniques were designed for sequential code and most would not be safe for multithreaded code. *How to prioritize multithreaded tests (with schedules)?* Test prioritization further improves on test selection: after we select to run a subset of tests, in which order should we run them to faster detect failures (if any)? Again, there are a number of techniques that showed good results for sequential code [10,11]. We plan to develop novel techniques for test selection and prioritization for multithreaded code. Our first step was on improving mutation testing for multithreaded code [5].

3. SOME RELATED WORK

Writing multithreaded unit tests in plain Java or C# has several problems due to reliance on real time to express specific schedules. Others also observed these problems and proposed specialized test frameworks, including ConAn [7], ConcJUnit [9], MultithreadedTC [8], and ThreadControl [4]. MultithreadedTC is the most related but has several differences: (1) It uses one, hard-coded policy for handling events such as blocking threads and advancing time; (2) It only allows abstract time to be observed in the test code and not in the code under test; and (3) Setting appropriate linear values for abstract time is tedious and error-prone.

Gambit [1] is an extension of the CHESS tool for performing context-bounded exploration of unit tests for concurrent libraries. Gambit allows two indirect ways to prioritize/select schedules: (1) It replaces depth-first search with best-first search, which uses a priority function to decide which branch/schedule to explore next; (2) User can specify prioritization and preemption sealing, namely which methods/variables are more important so that higher priority is assigned to exploring schedules involving those methods/variables. However, it does not consider code evolution.

4. REFERENCES

[1] T. Ball, S. Burckhardt, K. Coons, M. Musuvathi, and S. Qadeer. Preemption Sealing for Efficient Concurrency Testing. Technical report, Microsoft Research, 2009.

[2] F. Chen, P. Meredith, D. Jin, and G. Rosu. Efficient formalism-independent monitoring of parametric properties. In *ASE*, 2009.

[3] F. Chen, T. F. Şerbănuţă, and G. Roşu. jPredictor: A predictive runtime analysis tool for Java. In *ICSE*, 2008.

[4] A. Dantas, F. V. Brasileiro, and W. Cirne. Improving automated testing of multi-threaded software. In *ICST*, 2008.

[5] M. Gligoric, V. Jagannath, and D. Marinov. MuTMuT: Efficient exploration for mutation testing of multithreaded code. In *ICST*, 2010. (To appear.).

[6] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM TOSEM*, 2001.

[7] B. Long, D. Hoffman, and P. A. Strooper. Tool support for testing concurrent java components. *IEEE TSE*, 2003.

[8] W. Pugh and N. Ayewah. Unit testing concurrent software. In *ASE*, 2007.

[9] M. Ricken and R. Cartwright. ConcJUnit: Unit testing for concurrent programs. In *PPPJ*, 2009.

[10] G. Rothermel, R. J. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE TSE*, 2001.

[11] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA*, 2002.

[12] J. Zheng, B. Robinson, L. Williams, and K. Smiley. Applying regression test selection for COTS-based applications. In *ICSE*, 2006.