STORM: Static Unit Checking of Concurrent Programs[*]

Zvonimir Rakamarić
Department of Computer Science, University of British Columbia, Canada
zrakamar@cs.ubc.ca

ABSTRACT

Concurrency is inherent in today's software. Unexpected interactions between concurrently executing threads often cause subtle bugs in concurrent programs. Such bugs are hard to discover using traditional testing techniques since they require executing a program on a particular unit test (i.e. input) through a particular thread interleaving. A promising solution to this problem is static program analysis since it can simultaneously check a concurrent program on all inputs as well as through all possible thread interleavings. This paper describes a scalable, automatic, and precise approach to static unit checking of concurrent programs implemented in a tool called STORM. STORM has been applied on a number of real-world Windows device drivers, and the tool found a previously undiscovered concurrency bug in a driver from Microsoft's Driver Development Kit.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*assertion checkers, formal methods*

General Terms

Reliability, Verification

Keywords

Static Analysis, Concurrent Programs, Unit Checking

1. PROBLEM AND MOTIVATION

Today's software systems are prevalently concurrent and therefore hard to get right. Unexpected asynchronous interactions between concurrently executing threads often cause subtle bugs in concurrent programs. Such bugs are hard to discover using traditional testing techniques since they require executing a program on a particular unit test (i.e. input) through a particular thread interleaving, which creates a twofold problem. First, programmers are forced to write many unit tests to increase code coverage. Second, for each unit test the program under test has to be executed through as many interleavings as possible in the hope that an erroneous interleaving is going to be discovered. Typically, *stress-testing* is used when trying to produce interesting interleavings. This technique executes a program under heavy load (e.g. by creating many threads) over and over again hoping to produce different interleavings. However, in such a setting it is extremely difficult to determine which interleavings were actually executed and to measure coverage with respect to concurrency (i.e. interleavings). Furthermore, empirical evidence shows that the coverage with respect to all possible interleaving is still usually very small even after days of heavy stress-testing [6]. Both of these make *stress-testing* completely inadequate in the context of unit testing of concurrent programs.

A promising solution to this problem is static program analysis since it can simultaneously check a concurrent program on all inputs as well as through all possible thread interleavings. Context-bounded analysis is an attractive approach to verification of concurrent programs. This approach advocates analyzing all executions of a concurrent program in which the number of contexts executed per thread is bounded by a given constant K. Bounding the number of contexts executed per thread reduces the asymptotic complexity of checking concurrent programs: while reachability analysis of concurrent boolean programs is undecidable, the same analysis under a context-bound is NP-complete. Moreover, there is ample empirical evidence that synchronization errors, such as data races and atomicity violations, are manifested in concurrent executions with a small number of context switches [6, 7]. These two properties together make context-bounded analysis an effective approach for finding concurrency errors. At the same time, context-bounding provides a useful trade-off between the cost and coverage of verification. This paper describes how to employ context-bounded analysis in a scalable, automatic, and precise approach to static unit checking of concurrent programs. We implemented the approach in a tool called STORM and applied it on a number of real-world Windows device drivers.

2. BACKGROUND AND RELATED WORK

The idea of context-bounded analysis of concurrent programs was first proposed by Qadeer and Wu [7]. Many subsequent approaches have relied on bounding the number of contexts to tackle the complexity and scalability issues of concurrent program analysis. KISS [7] transforms a concurrent program with up to two context switches into a sequential one by mimicking context switches using procedure calls. However, restricting the number of context switches in such a way can be limiting. CHESS [6] is a tool for unit testing of multithreaded programs that dynamically explores thread

[*]This is a joint work with Alan J. Hu, Shuvendu Lahiri, and Shaz Qadeer. It was supported by a Microsoft Research Graduate Fellowship.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '10, May 2-8 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-719-6/10/05 ...$10.00.

interleavings by iteratively bounding the number of contexts. On the other hand, STORM is a static analysis tool and therefore does not have to execute the code and offers more coverage since it explores all possible paths in a program up to a given context bound.

3. UNIQUENESS OF THE APPROACH

In this work, we apply context-bounded verification to concurrent C programs such as those found in low-level systems code. In order to deal with the complexity of low-level concurrent C programs, we take a novel and unique three-step approach. First, we eliminate all the complexities of C, such as dynamic memory allocation, pointer arithmetic, casts, etc. by compiling into the Boogie programming language (BoogiePL) [2], a simple procedural language with scalar and map data types. Thus, we obtain a concurrent BoogiePL program from a concurrent C program. Second, we eliminate the complexity of concurrency by appealing to the recent method of Lal and Reps [5] for reducing context-bounded verification of a concurrent boolean program to the verification of a sequential boolean program. By adapting this method to the setting of concurrent BoogiePL programs, we are able to construct a sequential BoogiePL program that captures all behaviors of the concurrent BoogiePL program (and therefore of the original C program as well) up to the context-bound. Third, we generate a verification condition from the sequential BoogiePL program and check it using a Satisfiability Modulo Theories (SMT) solver.

In order to scale our verification to realistic C programs, we introduce the novel idea of *field abstraction*. The main insight is that the verification of a given property typically depends only on a small number of fields in the data structures of the program. Our algorithm partitions the set of fields into *tracked* and *untracked* fields; we only track accesses to the tracked fields and abstract away accesses to the untracked fields. This approach not only reduces the complexity of sequential code being checked, but also allows us to soundly drop context-switches from the program points where only untracked fields are accessed. Field abstraction is crucial for scalability of our verification technique.

4. RESULTS

This section describes STORM's tool flow, and our experience applying the tool on a number of real-life benchmarks. As described earlier, STORM first uses the HAVOC tool [3] to translate a multithreaded C program along with a set of relevant fields into a multithreaded BoogiePL program, then reduces it to a sequential BoogiePL program, and finally uses BOOGIE to check the sequential program. The BOOGIE verifier [2] generates a verification condition from the BoogiePL description using a variation of the standard weakest precondition transformer. Then, it employs the SMT solver Z3 [4] to check the resulting verification condition.

We evaluated STORM on a set of real-world Windows device driver benchmarks (see Table 1). We implemented a common unit test (i.e. harness) for putting device drivers through different concurrent scenarios. Each driver is checked in a scenario possibly involving concurrently executing driver dispatch routines, driver request cancellation and completion routines, and deferred procedure calls (column "Scenario"). The number of threads and the complexity of a scenario depend on the given driver's capabilities. For example, for the `usbsamp` driver, the unit test executes a dispatch, cancel, and completion routine in three threads. Apart from providing a particular scenario, our unit test also models synchronization provided by the device driver framework, as well as synchronization primitives, such as locks, that are used for driver-specific synchronization. STORM has the ability to check any user-specified

Driver	kLOC	Scenario	Time(s)
`usbsamp`	5	D | CA | CP	171
`ndis`	7	D | CA	13
`kbdclass`	7	D | CA	4
`mouclass`	7	D | CA	4
`pcidrv`	12	D | CA	18
`isousb`	12	D | CA | CP	81
`mqueue`	14	D | CA | CP | DPC	171
`daytona`	22	D | CA	2
`serial`	33	D | CA	133

Table 1: Experimental results on Windows device drivers. "kLOC" is the total number of lines of code in a driver; "Scenario" shows the concurrent unit test scenario being checked, i.e. which driver routines are executed concurrently as threads (D – dispatch routine, CA – cancel routine, CP – completion routine, DPC – deferred procedure call); "Time" is the running time of STORM for a given unit test scenario (in seconds).

safety property. In our experiments, we checked the *use-after-free* property for the `IRP` (*IO Request Packet*) data structure used by the device drivers.

Table 1 shows the result when the number of contexts per thread is up to 2. All experiments were conducted on an Intel Core2Duo at 3GHz running Windows 7, and all runtimes are in seconds. STORM managed to successfully check all of our benchmarks, which clearly demonstrates the scalability of our approach. In the process, STORM discovered a bug in the `usbsamp` driver from Microsoft's Driver Development Kit. It is important to note that this bug has not been found before by extensively applying other software checkers on `usbsamp`. For instance, SLAM [1] failed to discover this bug since SLAM can check only sequential code. KISS, on the other hand, can check concurrent code, but only up to 2 context switches. This bug occurs only after at least 3 context switches and therefore was missed by KISS as well.

5. REFERENCES

[1] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Intl. Symp. on Formal Methods for Components and Objects (FMCO)*, pages 364–387, 2005.

[3] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 19–33, 2007.

[4] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[5] A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *Intl. Conf. on Computer Aided Verification (CAV)*, pages 37–51, 2008.

[6] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 446–455, 2007.

[7] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 14–24, 2004.