

The Case for Context-Bounded Verification of Concurrent Programs

Shaz Qadeer
Microsoft Research
qadeer@microsoft.com

April 13, 2008

Concurrent programs are difficult to get right. Subtle interactions among communicating threads in the program can result in behaviors unexpected to the programmer. These behaviors typically result in bugs that occur late in the software development cycle or even after the software is released. Such bugs are difficult to reproduce and difficult to debug. As a result, they have a huge adverse impact on the productivity of programmers and the cost of software development. Therefore, tools that can help detect and debug concurrency errors will likely provide a significant boost to software productivity.

The problem of automatic and precise defect-detection for programs, whether concurrent or sequential, is undecidable in general. The importance of this problem has led researchers, over the past 50 years, to devise various techniques to circumvent this undecidability barrier. An important breakthrough in this direction is the idea of analyzing models of a program rather than the program itself. The fundamental insight is that although the set of behaviors of a concrete program might be insurmountably large, there is usually a simple abstract model of the program that contains enough information to prove that the program satisfies a particular partial specification. If the problem of analyzing the abstract model is decidable, then analysis of the model could help in pinpointing defects in the program. Of course, the model itself cannot be created automatically in general; algorithms for creating these models typically require input from the programmer.

Two models have been widely and successfully used for defect detection in sequential programs—finite-state machines and pushdown machines. For these models, the basic analysis problem, known as the *reachability* problem, is the following:

Given an error state e , is there an execution of the machine from the initial state to e ?

For both models, there are polynomial-time algorithms for solving the reachability problem; the complexity is linear for finite-state machines and cubic for pushdown machines. A big reason for the success of these models in software

verification is the relatively low complexity of reachability analysis for them. Unfortunately, the same modeling tools have not worked so well for concurrent programs. The natural extensions of these models to handle concurrency are communicating finite-state machines and communicating pushdown machines respectively. While reachability analysis is PSPACE-complete for communicating finite-state machines, it is undecidable for communicating pushdown machines. Consequently, the use of abstract models for verifying concurrent programs has seen limited success.

This position paper argues that the high complexity of reachability analysis for models of concurrent computation can be mitigated by performing context-bounded verification of these models. To understand the idea of context-bounded verification, we take a closer look at concurrent behaviors. We model a concurrent computation as an interleaving of actions performed by tasks that are concurrently executing and communicating using shared resources. A context switch occurs in such a computation whenever the execution of a task is temporarily interrupted by some other task. The *context-bounded reachability* problem, for both communicating finite-state machines and communicating pushdown machines, is the following:

Given an error state e and a bound $c \geq 0$, is there an execution of the machine from the initial state to e with no more than c context switches?

Note that context-bounded reachability is significantly different from depth-bounded reachability, where executions are restricted to some length $d \geq 0$. Bounding the number of context switches in an execution does not bound the length of the execution because a task may perform an unbounded number of steps prior to a context switch.

Bounding the number of context switches has a significant impact on the complexity of the reachability problem; context-bounded reachability is NP-complete both for communicating finite-state machines and communicating pushdown machines [5, 3]. Since the problem is NP-complete, it is unlikely that there is a polynomial-time algorithm for it. However, there is reason to be optimistic. In the last decade, significant advances have been made in satisfiability solving. Researchers have built a number of powerful satisfiability solvers that have managed to solve practical and real-world instances of hardware and software verification problems. Since context-bounded reachability is in NP, it can be translated to satisfiability thereby allowing these powerful solvers to be leveraged. In addition, there exist algorithms for solving context-bounded reachability that are polynomial in the size of the model and exponential in c [5, 2]. These algorithms could be used for scalable verification, at least for small values of c .

If an oracle for the context-bounded reachability problem returns the answer **No** for a particular bound c , then we are assured that the error state is unreachable via executions with up to c context switches. However, this answer says nothing about executions with more than c context switches. To decrease the likelihood of missed errors, we could start with a small value of c (zero, for

example) and keep incrementing it as long as the oracle returns the answer No, until the validation resources allocated to the program are exhausted. There are two advantages of this pay-as-you-go approach to verification. First, successful context-bounded verification for a bound c is a useful and intuitive coverage metric suitable for concurrent programs and orthogonal to sequential coverage metrics such as line or branch coverage. Second, we believe that the number of context switches in an execution is a good metric of the complexity of that execution. Hence, an erroneous execution returned by this iterative approach is one of the simplest witnesses to the error and could help the programmer localize the cause of the error quickly.

Obviously, the pay-as-you-go approach to verification of concurrent programs is just as easily facilitated by depth-bounding as by context-bounding. However, depth-bounding lacks the two aforementioned advantages of context-bounding because the depth of explored executions is not an intuitive metric for the complexity of concurrent executions.

The above discussion provides motivation for context-bounded verification from the point of view of computational complexity. However, our argument would rightly be considered weak if errors in concurrent programs manifested only in executions with a large number of context switches. Fortunately, there is significant empirical evidence indicating that is not the case. Over the past few years, we have implemented context-bounded reachability analysis in three software model checkers—KISS [6], ZING [1], and CHESS [4]. We have applied these tools to many real-world concurrent programs and discovered numerous errors exhibited by executions with a small number of context switches.

Context-bounding is a novel, interesting, and useful perspective on the problem of verifying concurrent systems. Recent work has provided both theoretical and practical evidence of the power of context-bounded verification. However, there are many important challenges ahead. First, to apply context-bounded verification to a concurrent program requires the identification of program tasks. In many programs, such as multithreaded programs manipulating shared data structures, the task abstraction is obvious and corresponds to the syntactic abstraction of a thread. However, for message-passing or event-driven programs, the task abstraction is not easily discerned. Consequently, we need linguistic techniques to specify and analysis techniques to discover tasks in concurrent programs. Second, to make use of satisfiability solvers we need efficient encodings of the context-bounded reachability problem into the satisfiability problem. Finally, we need techniques to construct concurrent finite-state and pushdown models from software implementations; these techniques must interact well with the encoding techniques.

References

- [1] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. Technical Report MSR-TR-2004-10, Microsoft Research, 2004.

- [2] Akash Lal and Thomas Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *CAV 08: Computer Aided Verification*, 2008.
- [3] Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas Reps. Interprocedural analysis of concurrent programs under a context bound. In *TACAS 08: Tools and Algorithms for the Construction and Analysis of Systems*, 2008.
- [4] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI 07: Programming Language Design and Implementation*, pages 446–455, 2007.
- [5] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *TACAS 05: Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer-Verlag, 2005.
- [6] Shaz Qadeer and Dinghao Wu. KISS: Keep it simple and sequential. In *PLDI 04: Programming Language Design and Implementation*, pages 14–24. ACM Press, 2004.