QED: A Proof System based on Reduction and Abstraction for the Static Verification of Concurrent Software

Tayfun Elmas
Koc University, College of Engineering, Istanbul, Turkey
telmas@ku.edu.tr

ABSTRACT

We present a proof system and supporting tool, QED, for the static verification of concurrent software. Our key idea is to simplify the verification of a program by rewriting it with larger atomic actions. We demonstrated the simplicity and effectiveness of our approach on benchmarks with intricate synchronization.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification — assertion checkers, correctness proofs, formal methods; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs — assertions, invariants, mechanical verification

General Terms

Languages, Theory, Verification

Keywords

Concurrent Programs, Atomicity, Reduction, Abstraction

1. PROBLEM AND MOTIVATION

Because of emerging technologies such as multicore processors and grid computing, concurrency is becoming an important issue of today's software systems and is likely to become even more so in the future. A wide-range of systems, e.g., web servers, databases, and operating systems, contain highly concurrent data structures and services in order to respond efficiently to a large number of clients accessing simultaneously. Such software makes use of sophisticated synchronization techniques, including fine-grained locking and non-blocking operations, and creates extra threads for internal operations. These techniques require intensive care to use, and bugs due to incorrect use of them can have serious consequences, such as data corruption, operating system crash, or even more catastrophic results, e.g., failure of an aircraft flight control system. Therefore, the functional correctness of software is as fundamental as its performance.

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.

This research focuses on statically verifying the (partial) functional correctness of concurrent software. The correctness is specified using two criteria: *assertion*s and *linearizability*. Assertions state local conditions expected to hold when the execution of the program reaches a certain location in the code. Linearizability formalizes the requirement that a concurrently-accessed data structure implementation conform to a specification containing atomic versions of the data structure's operations [9]. We provide a formal proof system and supporting software tool to show that 1) no execution of a program leads to an assertion violation or 2) every concurrent execution of a data structure is equivalent to an execution of its atomic specification.

2. BACKGROUND AND RELATED WORK

Because the full precise static verification is undecidable, the verification of a program is decomposed into decidable pieces, requiring code annotations (also called contracts) from the user. Thus, the effectiveness of the proof is highly dependent on the (manual) selection of these annotations. Annotations for concurrent programs require significantly more intellectual effort to state than those for sequential programs. The fundamental reason behind this is the interaction between threads over the shared memory: while writing the annotations for a program under fine-grained concurrency, one has to consider possible interleavings of a large number of conflicting operations.

In the invariant-based approaches, e.g. Owicki-Gries [13], each potential interleaving point in a program must be annotated with an invariant that is valid under interference from other concurrently-executing actions. Rely-guarantee methods [10] make this approach more modular by obviating the need to consider each pair of concurrent statements separately. Both these methods require the programmer to reason about interleavings of fine-grained actions; consequently, the annotations are complex. Concurrent separation logic [12] has the ability to maintain separation between shared and local memory, enabling sequential reasoning for multithreaded programs; however, this is not particularly useful for programs with high level of interference.

Fine-grained concurrency also complicates establishing an abstraction map and identifying the commit points of operations while proving linearizability [9]. Managing these requirements when the operations of a data structure are written in terms of many small actions that make visible changes to the state requires considerable expertise.

Atomicity is a well-known specification for concurrently executed code blocks. Several verification approaches were

developed to verify atomicity [6], using reduction as a key ingredient. Their use of reduction is limited to simple synchronization disciplines and can only reason about commutativity of accesses that are not simultaneously enabled. They require using auxiliary variables and access predicates to enable wider application of reduction. Abstractions have been used as a mechanism to prove atomicity in the work on purity [7].

On the other hand, our approach uses atomicity as a reasoning tool to enable more tractable verification of other specifications such as linearizability. Moreover, we use more flexible notions of reduction and abstraction. Our method is orthogonal and complementary to existing methods that do not make direct use of reduction and abstraction, by enhancing their applicability, and subsumes others that do.

3. APPROACH AND UNIQUENESS

The primary contribution of this work is a static proof system, called QED, for proving assertions [4] and linearizability [3]. QED provides a novel proof strategy in which *atomicity is used as a proof tool*: a program with fine-grained concurrency is transformed iteratively to make it consist of larger atomic actions, and the correctness conditions are checked after the program reaches an atomicity level appropriate for local analyses. This gradually reduces the influence of thread interleavings on the complexity of the reasoning, and therefore permits significantly more tractable proofs than those provided by existing methods. For example, we prove assertions by performing *sequential (local)* checks within atomic blocks of the final program. The transformations preserve or expand the set of behaviors of the program, so that assertions proved at the end of a sequence of transformations are valid in the original program. In addition, one can simplify a program with larger atomic blocks using QED and can continue the proof with another method, e.g., separation logic [12].

A distinguishing and essential aspect of our method is the iterative and alternating application of two kinds of transformations, *abstraction* and *reduction*, which allows us to reach the desired level of atomicity even when there is apparent interference between threads. While reduction and abstraction have been studied in isolation in the literature, they are *symbiotic* in QED. Reduction [11] replaces a compound statement consisting of several atomic actions with a single atomic action if certain non-interference conditions hold and allows a subsequent abstraction step to summarize the entire calculation in that statement locally. Abstraction replaces an atomic action with a more relaxed atomic action allowing more behaviors, permitting a later application of reduction to reason that it does not interfere with other atomic actions and merge it with other actions.

4. RESULTS AND CONTRIBUTIONS

I have implemented our verification method in a software tool, also called QED. QED is open source and can be downloaded from **http://qed.codeplex.com**.

Our tool accepts as input a multithreaded program in QEDPL language and a proof script. QEDPL is an extension of the Boogie programming language [2] of Microsoft Research with concurrency constructs, e.g., thread creation. We are also developing a translator from Java to QEDPL. The proof script contains a sequence of proof commands. A proof command is used for one of two purposes: to transform the input program using abstraction, reduction or a combination of the two, or to provide a concise specification of the behavior of the current version of the program, including locking protocols and data invariants. The tool automatically generates the verification conditions justifying each step of the proof and verifies them using Z3 [1], a state-of-the-art SMT solver developed by Microsoft Research. After executing each step in the proof script, QED allows the user to examine the resulting program, intercept the proof, and give new commands.

We also provide a set of proof script templates. These templates document and mechanize proof idioms, a sequence of low-level proof rules applied for a common scenario, for example, indicating that a variable is always lock-protected. We presented the proof idioms for common synchronization mechanisms, such as mutex and reader/writer locks in [5].

We have evaluated QED by verifying a number of multithreaded programs with varying degree of synchronization complexity. These examples include programs using fine-grained locking and non-blocking data structures. We have found that the iterative approach embodied in QED provides a simple and convenient way of communicating to the verifier the programmer's understanding of and correctness arguments on the computation and synchronization in the program. Thus, the proofs in our method are invariably cleaner than the proofs based on existing approaches.

The proof script of a program serves as a *reproducible* (by our tool) documentation of its correctness. Therefore, QED provides a proof repository for a collection of concurrent software including the following purity benchmarks from [7], fine grained multiset and lock-coupling linked list implementations, and non-blocking stack, queue, deque, and readers/writer lock implementations [8].

5. REFERENCES

[1] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. *TACAS'08*.

[2] R. DeLine and K. R. M. Leino. Boogiepl: A typed procedural language for checking object-oriented programs. *Technical Report MSR-TR-2005-70. Microsoft Research*. 2005.

[3] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying the proof of linearizability with reduction and abstraction. *TACAS'10*.

[4] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. *POPL'09*.

[5] T. Elmas, A. Sezgin, S. Tasiran, and S. Qadeer. An annotation assistant for interactive debugging of programs with common synchronization idioms. *PADTAD'09*.

[6] C. Flanagan and S. Qadeer. Types for atomicity. *TLDI'03*.

[7] S. N. Freund and S. Qadeer. Exploiting purity for atomicity. *IEEE Trans. Softw. Eng.*. 2005.

[8] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann. 2008.

[9] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 1990.

[10] C. B. Jones. Development Methods for Computer Programs including a Notion of Interference. *PhD thesis. Oxford University*. 1981.

[11] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*. 1975.

[12] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. *POPL'04*.

[13] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*. 1976.