Atomizer: A Dynamic Atomicity Checker For Multithreaded Programs (Summary)

Cormac Flanagan
Department of Computer Science
University of California at Santa Cruz
Santa Cruz, CA 95064

Stephen N. Freund
Department of Computer Science
Williams College
Williamstown, MA 01267

Abstract

Ensuring the correctness of multithreaded programs is difficult, due to the potential for unexpected interactions between concurrent threads. We focus on the fundamental non-interference property of atomicity *and present a dynamic analysis for detecting atomicity violations. This analysis combines ideas from both Lipton's theory of reduction and earlier dynamic race detectors such as Eraser. Experimental results demonstrate that this dynamic atomicity analysis is effective for detecting errors due to unintended interactions between threads. In addition, the majority of methods in our benchmarks are atomic, supporting our hypothesis that atomicity is a standard methodology in multithreaded programming.*

1 The Need for Atomicity

Multiple threads of control are widely used in software development because they help reduce latency and provide better utilization of multiprocessor machines. However, reasoning about the correctness of multithreaded code is complicated by the nondeterministic interleaving of threads and the potential for unexpected interference between concurrent threads. Since exploring all possible interleavings of the executions of the various threads is clearly impractical, methods for specifying and controlling the interference between concurrent threads are crucial for the development of reliable multithreaded software. Much previous work on controlling thread interference has focused on *race conditions*, which occur when two threads simultaneously access the same data variable, and at least one of the accesses is a write [1].

Unfortunately, the absence of race conditions is not sufficient to ensure the absence of errors due to unexpected interference between threads. As a concrete illustration of this limitation, consider the following excerpt from the class `java.lang.StringBuffer`. All fields of a `StringBuffer` object are protected by the implicit lock associated with the object, and all `StringBuffer` methods should be safe for concurrent use by multiple threads.

Excerpt from `java.lang.StringBuffer`

```
public final class StringBuffer {

   public synchronized
         StringBuffer append(StringBuffer sb) {
     int len = sb.length();
     ...  // other threads may change sb.length()
     ...  // len does not reflect length of sb
     sb.getChars(0, len, value, count);
     ...
   }

   public synchronized int length() { ... }
   public synchronized void getChars(...)  { ...}
   ...
}
```

The `append` method shown above first calls `sb.length()`, which acquires the lock `sb`, retrieves the length of `sb`, and releases the lock. The length of `sb` is stored in the variable `len`. At this point, a second thread could remove characters from `sb`. In this situation, `len` is now *stale* and no longer reflects the current length of `sb`, and so the `getChars` method is called with an invalid `len` argument, and may throw an exception. Thus, `StringBuffer` objects cannot be safely used by multiple threads, even though the implementation is race-free.

To catch errors like this, we focus on a widely-applicable non-interference property called *atomicity*. A procedure (or code block) is atomic if for every (arbitrarily interleaved) program execution, there is an equivalent execution with the same overall behavior where the atomic procedure is executed serially, that is, the procedure's execution is not interleaved with actions of other threads. This non-interference guarantee reduces the challenging problem of reasoning about an atomic procedure's behavior in a *multithreaded* context to the simpler problem of reasoning about the procedure's *sequential* behavior. The latter problem is significantly more amenable to standard techniques such as manual code inspection, dynamic testing, and static analysis.

In addition, atomicity is a natural methodology for multithreaded programming, and experimental results indicate that many existing procedures and library interfaces already follow this methodology [3]. Finally, many synchronization errors can be detected as atomicity violations.

Although atomicity is a widely-applicable and fundamental correctness property of multithreaded code, standard testing techniques are inadequate to verify atomicity. Testing may discover a particular interleaving on which an atomicity violation results in erroneous behavior, but the exponentially-large number of possible interleavings makes obtaining adequate test coverage essentially impossible.

2 Checking Atomicity Dynamically

We present a dynamic analysis for detecting atomicity violations. For each code block annotated as being atomic, our analysis verifies that every execution of that code block does not interfere with other threads. Intuitively, this approach increases the coverage of traditional dynamic testing. Instead of waiting for a particular interleaving on which an atomicity violation causes erroneous behavior, such as a program crash, the checker actively looks for evidence of atomicity violations that may cause errors under other interleavings. Our approach synthesizes ideas from dynamic race detectors (such as Eraser's *Lockset* algorithm) and Lipton's theory of reduction [4], as described below. We refer the reader to the extended version of this paper for the complete details of our approach, as well as a discussion of related race condition and atomicity checking tools [3].

2.1 The Lockset Algorithm

In order to identify atomicity violations, our tool first identifies race conditions using a variant of Eraser's *Lockset* algorithm [5]. This algorithm tracks a *lockset* for each shared variable. This lockset contains all locks that have been consistently held on all accesses to that variable. Each lock set initially contains all locks, and on each shared variable access, the Lockset algorithm removes from the corresponding lockset all locks not held by the current thread. If the lockset for a variable becomes empty, then no lock consistently protects all accesses to that variable, and our analysis assumes that there may be a race condition on that variable. This information regarding race conditions is then used by the following reduction algorithm.

2.2 The Theory of Reduction

Our analysis leverages Lipton's theory of reduction to dynamically detect atomicity violations. The theory of reduction is based on the notion of right-mover and left-mover actions [4]. An action b is a *right-mover* if, for any execution where the action b performed by one thread is immediately followed by an action c of a concurrent thread, the actions b and c can be swapped without changing the resulting state. Conversely, an action c is a *left-mover* if whenever c immediately follows an action b of a different thread, the actions b and c can be swapped, again without changing the resulting state. We classify operations performed by a thread as (left or right) movers as follows:

Operation	Mover Status
lock acquire	right-mover
lock release	left-mover
access to protected data	both-mover
access to unprotected data	non-mover

Once a thread acquires a lock, no other thread may acquire or release it. Hence the acquire operation can be moved to the right of a step by a concurrent thread without changing the resulting state. Similarly, a release operation commutes to the left. An access (read or write) to a shared variable that is protected by a lock is a *both-mover* since no other thread can simultaneously access that variable. In contrast, an access to a variable on which there may be race conditions is a *non-mover* since other threads may concurrently access the same variable.

To illustrate how the classification of actions as various kinds of movers enables us to verify atomicity, consider the first execution trace in the diagram below. In this trace, a thread (1) acquires a lock m, (2) reads a variable x protected by that lock, (3) updates x, and then (4) releases m. The execution path of this thread is interleaved with arbitrary actions b_1, b_2, b_3 of other threads. Because the acquire operation is a right-mover and the write and release operations are left-movers, there exists an equivalent serial execution (with the same final state Σ_7) in which the operations of this path are not interleaved with operations of other threads, as illustrated by the following diagram. Thus the execution path is atomic.

Reduced execution sequence

$$\begin{tikzcd}
\Sigma_0 \arrow[r, "acq(m)"] \arrow[drr, dashed] & \Sigma_1 \arrow[r, "b_1"] \arrow[d, dashed] & \Sigma_2 \arrow[r, "t=x"] & \Sigma_3 \arrow[r, "b_2"] \arrow[drr, dashed] & \Sigma_4 \arrow[r, "x=t+1"] \arrow[dl, dashed] & \Sigma_5 \arrow[r, "b_3"] \arrow[dr, dashed] & \Sigma_6 \arrow[r, "rel(m)"] \arrow[dll, dashed] & \Sigma_7 \\
\Sigma_0 \arrow[r, "b_1"] & \Sigma'_1 \arrow[r, "acq(m)"] & \Sigma'_2 \arrow[r, "t=x"] & \Sigma'_3 \arrow[r, "x=t+1"] & \Sigma'_4 \arrow[r, "rel(m)"] & \Sigma'_5 \arrow[r, "b_2"] & \Sigma'_6 \arrow[r, "b_3"] & \Sigma_7
\end{tikzcd}$$

More generally, suppose an execution path through a method contains a sequence of right-movers, followed by at most one non-mover action and then a sequence of left-movers. Then this path can be *reduced* to an equivalent serial execution, with the same resulting state, where the path is executed without any interleaved actions by other threads. Our dynamic analysis verifies that every executed trace through each atomic method is reducible, and it reports warnings when irreducible paths are observed.

3 Implementation and Evaluation

We have developed an implementation, called the *Atomizer*, of the dynamic analysis outlined above. The Atomizer takes as input a multithreaded Java program and rewrites the program to include additional instrumentation code. This instrumentation code calls appropriate methods of the Atomizer run-time library that implement the Lockset and reduction algorithms and issue warning messages when atomicity violations are detected.

For the `StringBuffer` class, the Atomizer detects that `append` contains a window of vulnerability between where the lock `sb` is released inside `length` and then re-acquired inside `getChars`, and produces the following warning, even on executions where this window of vulnerability is not exploited to produce an observable error.

Error report

```
StringBuffer.append is not atomic:
Atomic block entered
  at StringBuffer.append(StringBuffer.java:445)
  at BreakStringBuffer.main(Test.java:21)

Atomic block commits at lock release:
  at StringBuffer.length(StringBuffer.java:144)
  at StringBuffer.append(StringBuffer.java:451)
  at Test.main(Test.java:21)

Atomicity violation at lock acquire:
  at StringBuffer.getChars(StringBuffer.java:326)
  at StringBuffer.append(StringBuffer.java:455)
  at Test.main(Test.java:21)
```

The application of the Atomizer to over 100,000 lines of Java code demonstrates that it detects defects in multithreaded programs that would be missed by existing race-detection tools, and it produces fewer false alarms on benign races that do not cause atomicity violations. In addition, the Atomizer found no atomicity violations in over 90% of the methods annotated as atomic that were exercised during our test runs. While certainly sensitive to the coverage of our testing, this statistic suggest that atomicity is a fundamental design principle in many multithreaded systems, especially library classes and reusable application components.

4 Conclusions

Developing reliable multithreaded software is notoriously difficult, because concurrent threads often interact in unexpected and erroneous ways. Clearly, the cost-effective development of reliable multithreaded systems requires the development and application of methods for controlling the interference between concurrent threads. The notion of *atomicity* provides a strong (indeed maximal) and widely-applicable non-interference guarantee. This paper presents a dynamic analysis designed to catch atomicity violations would be missed by traditional testing or (static or dynamic) race-detection techniques.

We suggest that the wider adoption and emphasis on atomicity in multithreaded software could provide many benefits, which may include: simpler procedure specifications; better static analyses; decreased testing cost; easier code inspection; and detecting more scheduler-dependent bugs.

In additon, whereas this work has focused on multi-threaded systems, an interesting avenue for future work is to study what notions of non-interference similar to atomicity are appropriate for distributed systems.

Acknowledgments. This work was partly supported by the National Science Foundation under Grants CCR-0341179 and CCR-0341387, and by faculty research funds granted by the University of California at Santa Cruz and by Williams College.

References

[1] A. D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center, 1989.

[2] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 256–267, 2004.

[3] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[4] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.