Atomizer: A Dynamic Atomicity Checker For Multithreaded Programs

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. Much previous work has focused on detecting race conditions, but the absence of race conditions does not by itself prevent undesired thread interactions. We focus on the more fundamental non-interference property of *atomicity*; a method is atomic if its execution is not affected by and does not interfere with concurrently-executing threads. Atomic methods can be understood according to their sequential semantics, which significantly simplifies (formal and informal) correctness arguments.

This paper presents a dynamic analysis for detecting atomicity violations. This analysis combines ideas from both Lipton's theory of reduction and earlier dynamic race detectors. Experience with a prototype checker for multithreaded Java code demonstrates that this approach is effective for detecting errors due to unintended interactions between threads. In particular, our atomicity checker detects errors that would be missed by standard race detectors, and it produces fewer false alarms on benign races that do not cause atomicity violations. Our experimental results also indicate that the majority of methods in our benchmarks are atomic, supporting our hypothesis that atomicity is a standard methodology in multithreaded programming.

This is an updated version of our POPL'04 paper that correctly handles nested atomic blocks.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification–*reliability*; D.2.5 [Software Engineering]: Testing and Debugging–*monitors, testing tools*; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs.

General Terms: Languages, Algorithms, Verification.

Keywords: Atomicity, dynamic analysis, reduction.

1 Reliable Threads

Multiple threads of control are widely used in software development because they help reduce latency, increase throughput, and provide better utilization of multiprocessor machines. However, reasoning about the behavior and correctness of multithreaded code is difficult, due to the need to consider all possible interleavings of the executions of the various threads. Thus, methods for specifying and controlling the interference between threads are crucial to the cost-effective development of reliable multithreaded software.

Much previous work on controlling thread interference has focused on *race conditions*. A race condition occurs when two threads simultaneously access the same data variable, and at least one of the accesses is a write [47]. In practice, race conditions are commonly avoided by protecting each data structure with a lock [6]. This lock-based synchronization discipline is supported by a variety of type systems [21, 20, 19, 22, 8, 7, 28] and other static [49, 23, 10, 13] and dynamic [47, 11, 51, 42, 45] analyses.

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(),
     ...  // so len does not reflect the 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* [9] 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 free of race conditions.

Recent results have shown that subtle defects of a similar nature are common, even in well-tested libraries [24]. Havelund reports finding similar errors in NASA's Remote Agent spacecraft controller [1], and Burrows and Leino [9] and von Praun and Gross [51] have detected comparable defects in Java applications. Clearly, the construction of reliable multithreaded software requires the development and application of more systematic methods for controlling the interference between concurrent threads.

This paper focuses on a strong yet widely-applicable non-interference property called *atomicity*. A method (or in general a code block) is atomic if for every (arbitrarily interleaved) program execution, there is an equivalent execution with the same overall behavior where the atomic method is executed serially, that is, the method's execution is not interleaved with actions of other threads.

Atomicity corresponds to a natural programming methodology, essentially dating back to Hoare's monitors[1] [32]. Many existing classes and library interfaces already follow this methodology, our experimental results indicate that the vast majority of methods in our benchmarks are atomic.

In addition, atomicity provides a strong, indeed maximal, guarantee of non-interference between threads. This guarantee reduces the challenging problem of reasoning about an atomic method's behavior in a *multithreaded* context to the simpler problem of reasoning about the method's *sequential* behavior. The latter problem is significantly more amenable to standard techniques such as manual code inspection, dynamic testing, and static analysis.

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

This paper presents 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 is not affected by and 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 (described in Section 3.1). For the StringBuffer class described above, our technique 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 by concurrent threads.

[1]Monitors are less general in that they rely on syntactic scope restrictions and do not support dynamically-allocated shared data.

Atomizer error report

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

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

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

We have implemented this dynamic analysis in an automatic checking tool called the *Atomizer*. The application of this tool to over 100,000 lines of Java code demonstrates that it provides an effective approach for detecting defects in multithreaded programs, including some defects that would be missed by existing race-detection tools. In addition, the Atomizer produces fewer false alarms on benign races that do not cause atomicity violations. Finally, our results suggest that a large majority of the exported methods in our benchmarks are atomic, which validates our hypothesis that atomicity is a widely-used programming methodology.

We propose that the application of this technique during the development, validation, and evolution of multithreaded programs will provide multiple benefits, including:

- detecting atomicity violations that are resistant to both traditional testing and existing race detection tools,
- facilitating safe code re-use in multithreaded settings by validating atomicity properties of interfaces,
- simplifying code inspection and debugging, since atomic methods can be understood according to their sequential semantics, and
- improving concurrent programming methodology by encouraging programmers to document the atomicity guarantees provided by their code.

Dynamic atomicity checking complements existing static techniques, such as the type system for atomicity developed by Flanagan and Qadeer [24], since most software is currently validated using a combination of static type checking and dynamic testing. For large programs, a benefit of the dynamic approach is that it avoids the overhead of type annotations or type inference, particularly for legacy code. Combining dynamic atomicity checking with other static checkers for multithreaded code, such as the Calvin-R tool developed by Freund and Qadeer [26], would yield similar benefits.

The presentation of our results proceeds as follows. Section 2 introduces a model of concurrent programs that we use as the basis for our development. Section 3 describes our dynamic analysis for atomicity. Section 4 describes how the Atomizer implements this analysis, and Section 5 presents our experimental results. Section 6 discusses related work, and we conclude with Section 7.

2 Multithreaded Programs

To provide a formal basis for reasoning about interference between threads, we start by formalizing an execution semantics for multithreaded programs. In this semantics, a multithreaded program consists of a number of concurrently executing threads, each of which has an associated thread identifier $t \in \mathit{Tid}$. The threads communicate through a global store σ, which is shared by all threads. The global store maps program variables x to values v. The global store also records the state of each lock variable $m \in \mathit{Lock}$. If $\sigma(m) = t$, then the lock m is held by thread t; if $\sigma(m) = \bot$, then that lock is not held by any thread.

In addition to operating on the shared global store, each thread also has its own local store π containing data not manipulated by other threads, such as the program counter of that thread. A state $\Sigma = (\sigma, \Pi)$ of the multithreaded system consists of a global store σ and a mapping Π from thread identifiers t to the local store $\Pi(t)$ of each thread. Program execution starts in an initial state $\Sigma_0 = (\sigma_0, \Pi_0)$.

Domains

$$
\begin{array}{rrcl}
u, t \in & \mathit{Tid} & & \\
x \in & \mathit{Var} & & \\
v \in & \mathit{Value} & & \\
m \in & \mathit{Lock} & & \\
\sigma \in & \mathit{GlobalStore} & = & (\mathit{Var} \to \mathit{Value}) \cup (\mathit{Lock} \to (\mathit{Tid} \cup \{\bot\})) \\
\pi \in & \mathit{LocalStore} & & \\
\Pi \in & \mathit{LocalStores} & = & \mathit{Tid} \to \mathit{LocalStore} \\
\Sigma \in & \mathit{State} & = & \mathit{GlobalStore} \times \mathit{LocalStores}
\end{array}
$$

2.1 Standard semantics

We model the behavior of each thread in a multithreaded program as the transition relation T:

$$T \quad \subseteq \quad \mathit{Tid} \times \mathit{LocalStore} \times \mathit{Operation} \times \mathit{LocalStore}$$

The relation $T(t, \pi, a, \pi')$ holds if the thread t can take a step from a state with local store π, performing the operation $a \in \mathit{Operation}$ on the global store, yielding a new local store π'. The set of possible operations on the global store includes: $rd(x, v)$, which reads a value v from a variable x; $wr(x, v)$, which writes a value v to a variable x; $acq(m)$ and $rel(m)$, which acquire and release a lock m, respectively; $begin$ and end, which mark the beginning and end of an atomic block; and ϵ, the empty operation.

$$
\begin{array}{rcl}
a \in \mathit{Operation} ::= & & rd(x, v) \mid wr(x, v) \\
& \mid & acq(m) \mid rel(m) \\
& \mid & begin \mid end \mid \epsilon
\end{array}
$$

The following relation $\sigma \to_t^a \sigma'$ models the effect of an operation a by thread t on the global store σ. The global store $\sigma[x := v]$ is identical to σ except that it maps the variable x to the value v.

Effect of operations: $\sigma \to_t^a \sigma'$

[ACT READ]
$$\frac{\sigma(x) = v}{\sigma \to_t^{rd(x,v)} \sigma}$$

[ACT WRITE]
$$\frac{}{\sigma \to_t^{wr(x,v)} \sigma[x := v]}$$

[ACT OTHER]
$$\frac{a \in \{begin, end, \epsilon\}}{\sigma \to_t^{a} \sigma}$$

[ACT ACQUIRE]
$$\frac{\sigma(m) = \bot}{\sigma \to_t^{acq(m)} \sigma[m := t]}$$

[ACT RELEASE]
$$\frac{\sigma(m) = t}{\sigma \to_t^{rel(m)} \sigma[m := \bot]}$$

The following transition relation $\Sigma \to \Sigma'$ performs a single step of an arbitrarily chosen thread. We use $\to^*$ to denote the reflexive-transitive closure of $\to$. A transition sequence $\Sigma_0 \to^* \Sigma$ models the arbitrary interleaving of the various threads of a multithreaded program, starting from the initial state Σ_0. Although dynamic thread creation is not explicitly supported by the semantics, it can be modeled within the semantics in a straightforward way.

Standard semantics: $\Sigma \to \Sigma'$

[STD STEP]
$$\frac{T(t, \Pi(t), a, \pi') \qquad \sigma \to_t^a \sigma'}{(\sigma, \Pi) \to (\sigma', \Pi[t := \pi'])}$$

2.2 Serialized semantics

We assume the function $A : \mathit{LocalStore} \to \mathit{Nat}$ indicates the number of atomic blocks that are currently active, perhaps by examining the program counter and thread stack recorded in the local store. This count should be zero in the initial state, and should only change when entering or leaving an atomic block. We formalize these requirements as follows:

- $A(\Pi_0(t)) = 0$ for all $t \in \mathit{Tid}$;
- if $T(t, \pi, begin, \pi')$ then $A(\pi') = A(\pi) + 1$;
- if $T(t, \pi, end, \pi')$, then $A(\pi) > 0$ and $A(\pi') = A(\pi) - 1$;
- if $T(t, \pi, a, \pi')$ for $a \notin \{begin, end\}$, then $A(\pi) = A(\pi')$.

The relation $\mathcal{A}(\Pi)$ holds if any thread is inside an atomic block:

$$\mathcal{A}(\Pi) \quad \stackrel{\text{def}}{=} \quad \exists t \in \mathit{Tid}.\ A(\Pi(t)) \neq 0$$

The following *serialized* transition relation $\mapsto$ is similar to the standard relation $\to$, except that a thread cannot perform a step if another thread is inside an atomic block. Thus, the serialized relation $\mapsto$ does not interleave the execution of an atomic block with instructions of concurrent threads.

Serialized semantics: $\Sigma \mapsto \Sigma'$

[SERIAL STEP]
$$\frac{T(t, \Pi(t), a, \pi') \qquad \sigma \to_t^a \sigma' \qquad \forall u \neq t.\ A(\Pi(u)) = 0}{(\sigma, \Pi) \mapsto (\sigma', \Pi[t := \pi'])}$$

Reasoning about program behavior and correctness is much easier under the serialized semantics ($\mapsto$) than under the standard semantics ($\to$), since each atomic block can be understood sequentially, without the need to consider all possible interleaved actions of concurrent threads. However, standard language implementations only provide the standard semantics ($\to$), which admits additional transition sequences and behaviors. In particular, a program that behaves correctly according to the serialized semantics may still behave erroneously under the standard semantics. Thus, in addition to being correct with respect to the serialized semantics, the program should also use sufficient synchronization to ensure the atomicity of each block of code that is intended to be atomic. Thus, for any program execution $(\sigma_0, \Pi_0) \to^* (\sigma, \Pi)$ where $\neg\mathcal{A}(\Pi)$, there should exist an equivalent serialized execution $(\sigma_0, \Pi_0) \mapsto^* (\sigma, \Pi)$. We call this the *atomicity requirement* on program executions, and any execution of a correctly synchronized program should satisfy this requirement. (The restriction $\neg\mathcal{A}(\Pi)$ avoids consideration of partially-executed atomic blocks.)

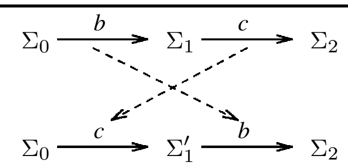
3 Dynamic Atomicity Checking

In this section, we present an instrumented semantics that dynamically detects violations of the above atomicity requirement. We start by reviewing Lipton's theory of reduction [38], which forms the basis of our approach.

3.1 Reduction

The theory of reduction is based on the notion of right-mover and left-mover actions. 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, as shown below:

Commuting actions



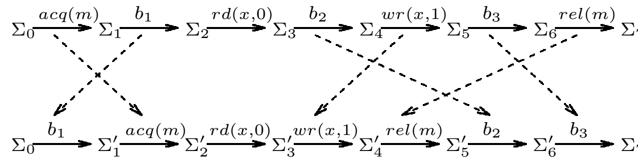
For example, if the operation b is a lock acquire, then the action c of the second thread neither acquires nor releases the lock, and so cannot affect the state of that lock. Hence the acquire operation can be moved to the right of c without changing the resulting state, and we classify each lock acquire operation as a right-mover.

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. Suppose the operation c by the second thread is a lock release. During b, the second thread holds the lock, and b can neither acquire nor release the lock. Hence the lock release operation can be moved to the left of b without changing the resulting state, and we classify each lock release operation as a left-mover.

Next, consider an access (read or write) to a variable that is shared by multiple threads. If the variable is protected by some lock that is held whenever the variable is accessed, then two threads can never access the variable at the same time, and we classify each access to that variable as a *both-mover*, which means that it is both a right-mover and a left-mover. If the variable is not consistently protected by some lock, we classify the variable access as a *non-mover*.

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 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



More generally, suppose a path through a code block 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.

3.2 Checking atomicity via reduction

We next leverage the theory of reduction to verify atomicity dynamically. In an initial presentation of our approach, we assume the programmer provides a partial function

$$P : \mathit{Var} \rightharpoonup \mathit{Lock}$$

that maps protected shared variables to associated locks; if $P(x)$ is undefined, then x is not protected by any lock.

We develop an instrumented semantics that only admits code paths that are reducible, and which goes wrong on irreducible paths. To record whether each thread is in the right-mover or left-mover part of an atomic block, we extend the state space with an *instrumentation store*:

$$\phi : \mathit{Tid} \rightarrow \{\mathit{InRight}, \mathit{InLeft}\}$$

Each state is now a triple (σ, ϕ, Π). If $A(\Pi(t)) \neq 0$, then thread t is inside an atomic block, and $\phi(t)$ indicates whether the thread is in the right-mover or left-mover part of that atomic block. The initial instrumentation store ϕ_0 is given by $\phi_0(t) = \mathit{InRight}$ for all $t \in \mathit{Tid}$.

The following relation $\Sigma \Rightarrow^a_t \phi'$ updates the instrumentation store whenever thread t performs operation a. The rule [INS ACCESS PROT] deals with an access to a protected variable while holding the appropriate lock. This action is a both-mover and so the instrumentation store ϕ does not change. Accesses to unprotected variables are non-movers, and they can occur outside atomic blocks: see [INS ACCESS OUTSIDE]. Unprotected accesses are also allowed inside an atomic block, and they cause a transition from the right-mover to the left-mover part of the atomic block: see [INS ACCESS COMMIT]. Acquire operations are right-movers, and they can occur outside or in the right-mover part of an atomic block, and conversely for release operations.

The relation $\Sigma \Rightarrow^a_t \mathit{wrong}$ holds if the operation a by thread t would *go wrong* by accessing a protected variable without holding the correct lock [WRONG RACE], or by performing a non-left-mover action in the left-mover part of an atomic block. Non-left-mover actions include accessing an unprotected variable [WRONG UNPROTECT] or acquiring a lock [WRONG ACQUIRE].

Instrumented operations: $\Sigma \Rightarrow_t^a \phi'$ **and** $\Sigma \Rightarrow_t^a \mathit{wrong}$

[INS ACCESS PROT]
$$\frac{\begin{array}{c} a \in \{rd(x,v), wr(x,v)\} \\ P(x) \text{ defined} \\ \sigma(P(x)) = t \end{array}}{(\sigma,\phi,\Pi) \Rightarrow_t^a \phi}$$

[INS ACCESS COMMIT]
$$\frac{\begin{array}{c} a \in \{rd(x,v), wr(x,v)\} \\ P(x) \text{ undefined} \\ A(\Pi(t)) \neq 0 \qquad \phi(t) = \mathit{InRight} \end{array}}{(\sigma,\phi,\Pi) \Rightarrow_t^a \phi[t := \mathit{InLeft}]}$$

[INS ACCESS OUTSIDE]
$$\frac{\begin{array}{c} a \in \{rd(x,v), wr(x,v)\} \\ P(x) \text{ undefined} \qquad A(\Pi(t)) = 0 \end{array}}{(\sigma,\phi,\Pi) \Rightarrow_t^a \phi}$$

[INS ACQUIRE]
$$\frac{\begin{array}{c} \phi(t) = \mathit{InRight} \\ \text{or} \quad A(\Pi(t)) = 0 \end{array}}{(\sigma,\phi,\Pi) \Rightarrow_t^{acq(m)} \phi}$$

[INS RELEASE]
$$\frac{}{(\sigma,\phi,\Pi) \Rightarrow_t^{rel(m)} \phi[t := \mathit{InLeft}]}$$

[INS RE-ENTER]
$$\frac{A(\Pi(t)) \neq 0}{(\sigma,\phi,\Pi) \Rightarrow_t^{begin} \phi}$$

[INS ENTER]
$$\frac{A(\Pi(t)) = 0}{(\sigma,\phi,\Pi) \Rightarrow_t^{begin} \phi[t := \mathit{InRight}]}$$

[INS OTHER]
$$\frac{a \in \{end, \epsilon\}}{(\sigma,\phi,\Pi) \Rightarrow_t^a \phi}$$

[WRONG RACE]
$$\frac{\begin{array}{c} a \in \{rd(x,v), wr(x,v)\} \\ P(x) \text{ defined} \\ \sigma(P(x)) \neq t \end{array}}{(\sigma,\phi,\Pi) \Rightarrow_t^a \mathit{wrong}}$$

[WRONG UNPROTECT]
$$\frac{\begin{array}{c} a \in \{rd(x,v), wr(x,v)\} \\ P(x) \text{ undefined} \\ A(\Pi(t)) \neq 0 \qquad \phi(t) = \mathit{InLeft} \end{array}}{(\sigma,\phi,\Pi) \Rightarrow_t^a \mathit{wrong}}$$

[WRONG ACQUIRE]
$$\frac{A(\Pi(t)) \neq 0 \qquad \phi(t) = \mathit{InLeft}}{(\sigma,\phi,\Pi) \Rightarrow_t^{acq(m)} \mathit{wrong}}$$

The instrumented transition relation $\Sigma \Rightarrow \Sigma'$ performs an instrumented step of an arbitrary thread; and $\Sigma \Rightarrow \mathit{wrong}$ holds if a step from Σ could violate the synchronization discipline or the atomicity requirement.

Instrumented semantics: $\Sigma \Rightarrow \Sigma'$ **and** $\Sigma \Rightarrow \mathit{wrong}$

[INS STEP]
$$\frac{\begin{array}{c} T(t, \Pi(t), a, \pi') \\ \sigma \rightarrow_t^a \sigma' \\ (\sigma,\phi,\Pi) \Rightarrow_t^a \phi' \end{array}}{(\sigma,\phi,\Pi) \Rightarrow (\sigma',\phi',\Pi[t := \pi'])}$$

[INS WRONG]
$$\frac{\begin{array}{c} T(t, \Pi(t), a, \pi') \\ \sigma \rightarrow_t^a \sigma' \\ (\sigma,\phi,\Pi) \Rightarrow_t^a \mathit{wrong} \end{array}}{(\sigma,\phi,\Pi) \Rightarrow \mathit{wrong}}$$

The following theorems state that the instrumented semantics is identical to the standard semantics, except that the instrumented semantics records additional information and may go wrong. In addition, any instrumented execution that does not go wrong satisfies the atomicity requirement.

THEOREM 1 (EQUIVALENCE OF SEMANTICS).

1. *If* $(\sigma,\phi,\Pi) \Rightarrow^* (\sigma',\phi',\Pi')$*, then* $(\sigma,\Pi) \rightarrow^* (\sigma',\Pi')$.
2. *If* $(\sigma,\Pi) \rightarrow^* (\sigma',\Pi')$ *then* $\forall \phi$ *either*
   (a) $(\sigma,\phi,\Pi) \Rightarrow^* \mathit{wrong}$*, or*
   (b) $\exists \phi'$ *such that* $(\sigma,\phi,\Pi) \Rightarrow^* (\sigma',\phi',\Pi')$.

PROOF: By induction over derivations.

THEOREM 2 (INSTRUMENTED REDUCTION).
If $(\sigma_0,\phi_0,\Pi_0) \Rightarrow^* (\sigma,\phi,\Pi)$ *and* $\neg\mathcal{A}(\Pi)$ *then* $(\sigma_0,\Pi_0) \mapsto^* (\sigma,\Pi)$.

PROOF: See Appendix.

If the instrumented semantics admits a particular execution, then not only is that execution reducible, but many similar executions are also reducible. In particular, when an atomic block is being executed, the only scheduling decision that affects program behavior is when the commit operation (the transition from $\mathit{InRight}$ to InLeft) is scheduled. Scheduling decisions regarding when other operations in the atomic block are scheduled are irrelevant, in that they do not affect program behavior or reducibility. Hence, one test run under our instrumented semantics can simultaneously verify the reducibility of many executions of the standard semantics.

3.3 Inferring protecting locks

The instrumented semantics of the previous section relies on the programmer to specify protecting locks for shared variables. To avoid burdening the programmer, we next extend the instrumented semantics to infer protecting locks, using a variant of Eraser's *Lockset* algorithm [47]. We extend the instrumentation store ϕ to map each variable x to a set of *candidate locks* for x, such that these candidate locks have all been held on every access to x seen so far:

$$\phi \;:\; (\mathit{Tid} \rightarrow \{\mathit{InRight}, \mathit{InLeft}\}) \cup (\mathit{Var} \rightarrow 2^{\mathit{Lock}})$$

The initial candidate lock set for each variable is the set of all locks, that is, $\phi_0(x) = \mathit{Lock}$ for all $x \in \mathit{Var}$.

The relation $\Sigma \Rightarrow_t^a \phi'$ updates the extended instrumentation store whenever thread t performs operation a on the global store. The rule [INS2 ACCESS] for a variable access removes from the variable's candidate lock set all locks not held by the current thread. We use $H(t,\sigma)$ to denote the set of locks held by thread t in state σ:

$$H(t,\sigma) = \{m \in \mathit{Lock} \mid \sigma(m) = t\}$$

If the candidate lock set for a variable becomes empty, then all accesses to that variable should be treated as non-movers, but previous accesses may already have been incorrectly classified as both-movers. For example, if $\phi(\mathtt{x}) = \{\mathtt{m}\}$ when thread t enters the following function `double`, then the first access to x by thread t will be classified as a both-mover. If, at that point, an action of a concurrent thread causes $\phi(\mathtt{x})$ to become empty, the analysis will classify the second access to x by t as a non-mover, but will not reclassify the first access, and thus the analysis will fail to recognize that `double` may not be reducible.

```
/*# atomic */ void double() {
  synchronized (m) {
    int t = x;
    x = 2 * t;
  }
}
```

Thus, to ensure soundness, the lock inference semantics does not support unprotected variables, and instead requires every variable to have a protecting lock. If the candidate lock set becomes empty, then that state goes wrong, via [WRONG2 RACE].

Instrumented operations 2: $\Sigma \Rrightarrow_t^a \phi'$ **and** $\Sigma \Rrightarrow_t^a \mathit{wrong}$

[INS2 ACCESS]
$$\frac{\begin{array}{c} a \in \{rd(x,v), wr(x,v)\} \\ \phi(x) \cap H(t,\sigma) \neq \emptyset \end{array}}{(\sigma,\phi,\Pi) \Rrightarrow_t^a \phi[x := \phi(x) \cap H(t,\sigma)]}$$

[INS2 ACQUIRE]
$$\frac{\begin{array}{c} \phi(t) = \mathit{InRight} \\ \text{or} \quad A(\Pi(t)) = 0 \end{array}}{(\sigma,\phi,\Pi) \Rrightarrow_t^{acq(m)} \phi}$$

[INS2 ENTER]
$$\frac{A(\phi(t)) = 0}{(\sigma,\phi,\Pi) \Rrightarrow_t^{begin} \phi[t := \mathit{InRight}]}$$

[INS2 RE-ENTER]
$$\frac{A(\phi(t)) \neq 0}{(\sigma,\phi,\Pi) \Rrightarrow_t^{begin} \phi}$$

[INS2 RELEASE]
$$\frac{}{(\sigma,\phi,\Pi) \Rrightarrow_t^{rel(m)} \phi[t := \mathit{InLeft}]}$$

[INS2 OTHER]
$$\frac{a \in \{end, \epsilon\}}{(\sigma,\phi,\Pi) \Rrightarrow_t^a \phi}$$

[WRONG2 RACE]
$$\frac{\begin{array}{c} a \in \{rd(x,v), wr(x,v)\} \\ \phi(x) \cap H(t,\sigma) = \emptyset \end{array}}{(\sigma,\phi,\Pi) \Rrightarrow_t^a \mathit{wrong}}$$

[WRONG2 ACQUIRE]
$$\frac{\begin{array}{c} A(\Pi(t)) \neq 0 \\ \phi(t) = \mathit{InLeft} \end{array}}{(\sigma,\phi,\Pi) \Rrightarrow_t^{acq(m)} \mathit{wrong}}$$

The relation $\Sigma \Rrightarrow \Sigma'$ performs an instrumented step (with lock inference) of an arbitrarily chosen thread; the relation $\Sigma \Rrightarrow \mathit{wrong}$ describes states that go wrong.

Instrumented semantics 2: $\Sigma \Rrightarrow \Sigma'$ **and** $\Sigma \Rrightarrow \mathit{wrong}$

[INS2 STEP]
$$\frac{\begin{array}{c} T(t,\Pi(t),a,\pi') \\ \sigma \rightarrow_t^a \sigma' \\ (\sigma,\phi,\Pi) \Rrightarrow_t^a \phi' \end{array}}{(\sigma,\phi,\Pi) \Rrightarrow (\sigma',\phi',\Pi[t := \pi'])}$$

[INS2 WRONG]
$$\frac{\begin{array}{c} T(t,\Pi(t),a,\pi') \\ \sigma \rightarrow_t^a \sigma' \\ (\sigma,\phi,\Pi) \Rrightarrow_t^a \mathit{wrong} \end{array}}{(\sigma,\phi,\Pi) \Rrightarrow \mathit{wrong}}$$

Like the previous instrumented semantics ($\Rightarrow$), the lock-inference semantics ($\Rrightarrow$) is equivalent to the standard semantics ($\rightarrow$) except that it only admits execution sequences that satisfy the atomicity requirement. The following two theorems formalize these correctness properties. Their proofs are analogous to those of Theorems 1 and 2.

THEOREM 3 (EQUIVALENCE OF SEMANTICS 2).

1. *If* $(\sigma,\phi,\Pi) \Rrightarrow^* (\sigma',\phi',\Pi')$*, then* $(\sigma,\Pi) \rightarrow^* (\sigma',\Pi')$.
2. *If* $(\sigma,\Pi) \rightarrow^* (\sigma',\Pi')$ *then* $\forall\phi$ *either*
   (a) $(\sigma,\phi,\Pi) \Rrightarrow^* \mathit{wrong}$*, or*
   (b) $\exists\phi'$ *such that* $(\sigma,\phi,\Pi) \Rrightarrow^* (\sigma',\phi',\Pi')$.

THEOREM 4 (INSTRUMENTED REDUCTION 2).
If $(\sigma_0,\phi_0,\Pi_0) \Rrightarrow^* (\sigma,\phi,\Pi)$ *and* $\neg\mathcal{A}(\Pi)$ *then* $(\sigma_0,\Pi_0) \mapsto^* (\sigma,\Pi)$.

Again, if the instrumented semantics admits a particular execution, then all executions that are equivalent to that execution modulo irrelevant scheduling decisions are reducible.

4 Implementation

We have developed an implementation, called the *Atomizer*, of the dynamic analysis outlined in the previous section. The Atomizer takes as input a multithreaded Java [27] program and rewrites the program to include additional instrumentation code. This instrumentation code calls appropriate methods of the Atomizer runtime that implement the Lockset and reduction algorithms and issue warning messages when atomicity violations are detected.

The Atomizer performs the instrumentation on Java source code. This approach has a number of advantages: it supports programmer-supplied annotations; it works at the high level of abstraction of the Java language; and it is portable across all Java virtual machines. This approach does require source code, but the instrumentation could also be performed at the bytecode level.

The target program can include annotations in comments to indicate that a method is atomic, as in:

```
/*# atomic */ void getChars() {...}
```

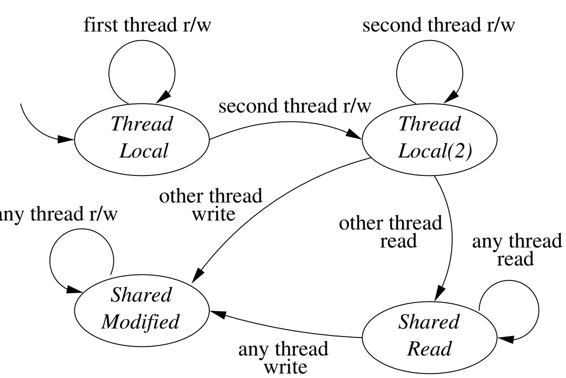
The Atomizer supports additional annotations to specify that a code block is atomic, to suppress spurious warnings, to ignore races on specific fields, and so on. Alternatively, the Atomizer can apply heuristics to decide which blocks should be checked for atomicity. These heuristics are that (1) all methods *exported* by classes should be atomic, and (2) all synchronized blocks and synchronized methods should be atomic. Exported methods are those that are public or package protected. However, these heuristics are not used for `main` and the `run` methods of `Runnable` objects, because these methods typically are not atomic. Although these heuristic are quite simple, they provide a reasonable starting point for identifying atomicity errors in unannotated code.

In the rest of this section, we describe our Lockset and reduction implementation, demonstrate how the tool identifies and reports errors, and present several improvements to the basic algorithm.

4.1 Lockset algorithm

For each field of each allocated object, the Atomizer tracks a *state* that reflects the degree to which the field has been shared among multiple threads.

Lockset algorithm states for each allocated field



The following possible states of our algorithm are similar to the states in earlier race detectors [47, 50]:

Thread-Local: The field has only been accessed by the object's creating thread.

Thread-Local (2): Ownership has transferred to a second thread, and the field is no longer accessed by the creating thread. This state supports common initialization patterns in Java [50].

Read-Shared: The field has been read, but not written, by multiple threads.

Shared-Modified: The field has been read and written by multiple threads, and a candidate lock set records which locks have been consistently held when accessing this field. When entering this state, the candidate set is initialized with all locks held by the current thread.

When a thread accesses a field, the Atomizer run-time updates its state according to the following transition diagram. (The Atomizer does not instrument array accesses.)

4.2 Reduction algorithm

The instrumented semantics for lock inference in Section 3.3 goes wrong on any race condition. Since programs frequently have benign races, the Atomizer implements a relaxed version of this semantics that accommodates such benign race conditions. If the candidate lock set for a variable becomes empty, then subsequent accesses to that variable are considered non-movers. Note that previous accesses to that variable, which were earlier classified as movers, will not be re-classified as non-movers, since storing a history of all variable accesses would be expensive. Thus, as mentioned in Section 3.3, these relaxed rules introduce a degree of unsoundness. We believe this unsoundness rarely causes the Atomizer to miss atomicity violations in practice because it requires an unlucky scheduling of operations and because the Atomizer will report the problem on the next execution of the non-atomic code fragment. The following rules adapt the relations $\Sigma \Rrightarrow^a_t \phi'$ and $\Sigma \Rrightarrow^a_t$ *wrong* to express this relaxed semantics.

Relaxed instrumentation: $\Sigma \Rrightarrow^a_t \phi'$ **and** $\Sigma \Rrightarrow^a_t$ *wrong*

[INS2 RACE]

$$\frac{\begin{array}{c} a \in \{rd(x,v), wr(x,v)\} \\ \phi(x) \cap H(t,\sigma) = \emptyset \\ A(\Pi(t)) = 0 \quad \text{or} \quad \phi(t) = \mathit{InRight} \end{array}}{(\sigma,\phi,\Pi) \Rrightarrow^a_t \phi[t := \mathit{InLeft}, x := \emptyset]}$$

[WRONG2 RACE]

$$\frac{\begin{array}{c} a \in \{rd(x,v), wr(x,v)\} \\ \phi(x) \cap H(t,\sigma) = \emptyset \\ A(\Pi(t)) \neq 0 \qquad \phi(t) = \mathit{InLeft} \end{array}}{(\sigma,\phi,\Pi) \Rrightarrow^a_t \mathit{wrong}}$$

To produce clear error messages like that in Section 1, the Atomizer can optionally capture stack traces (in the form of `Exception` objects) at the entry and commit points of each atomic block, and include these stack traces in error messages. Since the Atomizer supports nested atomic blocks, a single operation could result in multiple atomicity violations.

4.3 Extensions

The Atomizer may produce false alarms due to imprecisions in the Lockset and reduction algorithms. We next present several improvements that eliminate many of these false alarms. We start by revisiting the treatment of synchronization operations during reduction. The classification of lock acquires and releases as right-movers and left-movers, respectively, is correct but overly-conservative in some cases. In particular, modular programs typically include redundant synchronization operations that we can more precisely characterize as both-movers.

Re-entrant locks. Lock acquires are in general only right-movers and not left-movers. However, Java provides re-entrant locks, and a re-entrant lock acquire is a both-mover, because this operation cannot interact with other threads. Similarly, a re-entrant release is also a both-mover.

Thread-local locks. If a lock is used by only a single thread, acquires and releases of that lock are both-movers.

Thread-local (2) locks. Adding another *Thread-local* state, as in our Lockset algorithm, eliminates false alarms caused by initialization patterns in which one thread creates and initializes a protected object, and then transfers ownership of both the object and its protecting lock to another thread.

Protected locks. Suppose each thread always holds some lock m_1 before acquiring lock m_2. In this case, two threads cannot attempt to acquire m_2 simultaneously, and so operations on the lock m_2 are also both-movers.

Write-protected data. Consider the following two methods, in which the variable `x` is protected by a lock for all writes, but not protected for reads.

```
/*# atomic */ int read() { return x; }
/*# atomic */ void inc() {
    synchronized (lock) { x = x+1; }
}
```

If `x` is a 32-bit variable, then the `read()` method is atomic on a sequentially-consistent machine, even though no protecting lock is held. Despite the presence of such unprotected reads, the `inc()` method is also atomic. In particular, when the lock is held, a read of `x` is a both-mover, since no other thread can write to `x` without holding the lock.

To handle examples like this one, we use a variant of the Lockset algorithm. For each field, this algorithm infers a *lock set pair*, consisting of:

1. an *access-protecting* lock set, which contains locks held on every access (read or write) to that field, and
2. a *write-protecting* lock set, which contains locks held on every write to that field.

The access-protecting lock set is always a subset of the write-protecting lock set. A field read is a both-mover if the current thread holds at least one of the write-protecting locks; otherwise the read is a non-mover. In contrast, a field write is a both-mover only if the access-protecting lock set is non-empty; otherwise the write is a non-mover.

Using these lock set pairs, the Atomizer can infer that `inc()` is atomic, since it consists of a right-mover (the lock acquire); a both-mover (the read of `x`); an atomic action (since the write of `x` does not commute with concurrent reads of other threads); and a left-mover (the lock release). In comparison, existing race-detection tools would produce a false warning about the race condition in `read()`, even though this race condition is benign and does not affect the atomicity of either method.

Benchmark	Lines	Num. Threads	Num. Locks	Max. Locks Held	Num. Lock Set Pairs	Base Time (s)	Atomizer Slowdown	Atomicity Warnings	Errors
elevator	529	5	8	1	17	11.14	—	2	0
hedc	29,948	26	385	3	728	8.36	—	4	1
tsp	706	10	2	1	5	0.94	48.2	7	0
sor	17,690	4	1	1	2	0.70	7.3	0	0
moldyn	1,291	5	1	1	2	3.62	11.8	0	0
montecarlo	3,557	5	1	1	2	7.94	2.2	1	0
raytracer	1,859	5	5	1	7	5.96	36.6	1	1
mtrt	11,315	6	7	2	7	2.33	46.4	6	0
jigsaw	90,100	53	706	31	4,531	13.49	4.7	34	1
specJBB	30,490	10	262,000	6	340,088	18.01	11.2	4	0
webl	22,284	5	402,445	3	452,685	60.35	—	19	0
lib-java	75,305	39	816,617	6	986,855	96.5	—	19	4

Figure 1. Summary of test programs and performance.

5 Evaluation

This section summarizes our experience applying the Atomizer to twelve benchmark programs. These programs include `elevator`, a discrete event simulator for elevators [51]; `hedc`, a tool to access astrophysics data from Web sources [51]; `tsp`, a Traveling Salesman Problem solver [51]; `sor`, a scientific computing program [51]; `mtrt`, a multithreaded ray-tracing program from the SPEC JVM98 benchmark suite [48]; `jigsaw`, an open source web server [54] configured to serve a fixed number of pages to a crawler; `specJBB`, the SPEC JBB2000 business object simulator [48] configured to process a fixed number of transactions; `moldyn`, `montecarlo`, and `raytracer` from the Java Grande benchmark suite [33]; `webl`, a scripting language interpreter for processing web pages, configured to execute a simple web crawler [34]; and `lib-java`. This last program is an uninstrumented test harness (comprised of `webl`, `jbb`, and `hedc`) that tests an instrumented version of the standard Java libraries `java.lang`, `java.io`, `java.net`, and `java.util`. All programs other than `lib-java` use uninstrumented libraries.

The Atomizer instrumented these programs using the heuristics described in Section 4 (exported methods and synchronized blocks are annotated as atomic). To ensure that our measurements would accurately reflect the cost of the underlying analysis, the Atomizer did not record stack histories for atomic block entry and commit points for these tests. We performed the experiments on a Red Hat Linux 8.0 computer with dual 3.06GHz Pentium 4 Xeon processors and 2GB of memory. We used the Sun JDK 1.4.2 compiler and virtual machine for all benchmarks except `lib-java`, for which we used the Sun JDK 1.3.1 virtual machine due to compatibility problems.

Figure 1 presents statistics for the test programs using all the extensions from Section 4.3. The number of locks and distinct lock set pairs were relatively small for most programs, although the larger programs used many objects as locks, in some cases several orders of magnitude more than in comparably-sized C programs [47].

The slowdown incurred by the instrumentation varied from 2.2x to roughly 50x. We only report slowdowns for compute-bound programs. Those programs with very little slowdown, such as `sor` and `montecarlo`, spent most of the time in uninstrumented library code. We believe that slowdowns of 20x–40x are representative for most programs. However, we did not focus on efficiency in this prototype, and there is much room for improvement. In particular, static analyses have reduced the overhead of dynamic race detection to under 50% [51], which suggests that similar performance could be achieved when checking atomicity.

The "Atomicity Warnings" column in Figure 1 reports the number of atomic blocks and methods that failed the Atomizer's atomicity requirements during test runs. Figure 2 shows the cumulative benefit of the extensions from Section 4.3. The "Basic" column indicates the number of warnings reported for each program using the basic Lockset and reduction algorithms. The succeeding columns show the number of warnings as each refinement from Section 4.3 is added. Cumulatively, these five refinements are quite effective: they reduce the number of warnings by roughly 70% (from 341 to 97).

The last column in Figure 1 reports the number of atomicity violations that we consider errors, either because they could lead to undesirable program behavior or because they violate documented atomicity properties. Despite checking only mature software, the Atomizer identified a number of potentially damaging errors. Half of the errors were reported for atomic blocks with multiple data races or a single data race followed by a lock acquire. The remainder contained a lock acquire operation after a lock release.

When testing the instrumented libraries, the Atomizer warned of an atomicity violation in the synchronized method `PrintStream.println(String s)`, which uses two calls to write the string `s` and the following new-line character to a stream stored in the instance variable `out`. However, a different thread in the system also wrote to `out`, potentially at the same time, which could cause the output stream to be corrupted. A comparable error in `PrintWriter` had been previously identified by a static type system [24], but the Atomizer caught this defect with no programmer intervention and pinpointed an exact location in the program where the bug could manifest itself.

The Atomizer reported an error in `jigsaw` that was also found statically using a *view consistency* analysis [51]. In this case, a specific interleaving could allow an entry to be added to a resource store after the store had been closed as part of the shutdown process. Other errors included a known problem with `Hashtable` iterators in the presence of concurrent modifications, and a case where multiple threads updated a `Calendar` object through non-atomic methods.

In most programs, the warnings that did not indicate defects could be suppressed by inserting a handful of annotations. A significant number of false alarms were due to the overly-optimistic heuristics employed by the Atomizer to identify atomic blocks. Fewer false alarms would be produced when checking code with programmer-inserted `/*# atomic */` declarations. For example, atomicity violations were often reported on methods called near the top-level

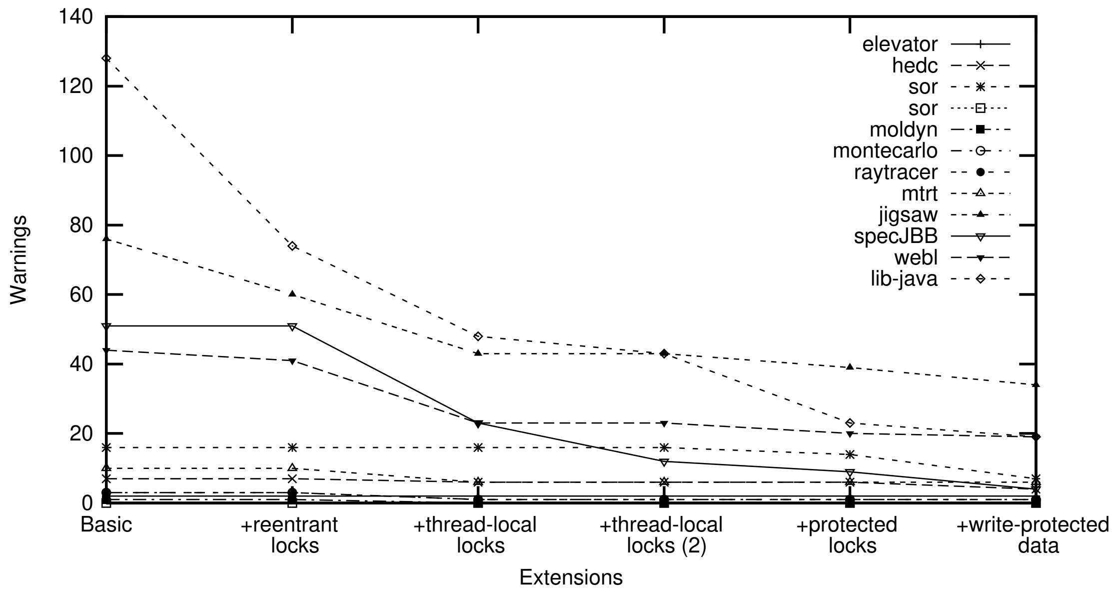


Figure 2. Warnings reported by the Atomizer under different configurations.

entry points of the program (the `main` and `run` methods), but many such methods are not intended to be atomic and would not be labeled as atomic by a programmer. Other common sources of false alarms include double-checked locking patterns, lazy initialization patterns, and various caching idioms. These programming idioms are notoriously problematic for analysis tools based on race detection and are discussed in more detail in [42]. Although some of these practices, such as double-checked locking, are incompatible with the Java memory model specification, we classify them as false alarms since they do not cause problems in most current Java environments [42].

During these tests, the Atomizer also recognized five fields with benign race conditions that did not lead to atomicity violations. In these cases, the Atomizer did not report spurious warnings to the user, as would have been the case for race condition checkers.

Overall, the Atomizer found no potential atomicity violations in over 90% of the methods annotated as atomic that were exercised during our test runs. These statistics suggest that atomicity is a fundamental design principle in many multithreaded systems, especially library classes and reusable application components.

6 Related Work

Lipton [38] first proposed reduction as a way to reason about concurrent programs without considering all possible interleavings. He focused primarily on checking deadlock freedom. Doeppner [15], Back [4], and Lamport and Schneider [37] extended this work to allow proofs of general safety properties. Cohen and Lamport [12] extended reduction to allow proofs of liveness properties. Misra [41] has proposed a reduction theorem for programs built with monitors [32] communicating via procedure calls.

Eraser [47] introduced the Lockset algorithm for dynamic race detection. This approach has been extended to object-oriented languages [50] and has been improved for precision and performance [11, 45]. O'Callahan and Choi [42] recently combined the Lockset algorithm with a happens-before analysis to reduce false alarms in a dynamic race detector for Java programs.

A number of static race detectors have also been developed. Warlock [49] is a static race detection system for C programs. ESC/Java [23] statically catches a variety of software defects, including race conditions. Other approaches for static race and deadlock prevention are discussed in earlier papers [20, 19, 21]; these include model-checking [10, 13, 18], dataflow analysis [16], abstract interpretation [44], and type systems for process calculi [35, 36].

In previous work, we produced a type system [21] that prevents violations of the lock-based synchronization discipline. Since then, similar type systems have been developed that include a notion of object ownership [8], and that target other languages such as Cyclone [28], a type-safe variant of C. Compared to dynamic techniques, these static type systems provide stronger soundness guarantees and detect errors earlier in the development cycle, but require more effort from programmer.

While some of these race detection tools have been quite effective, they may fail to detect atomicity violations and may yield false alarms on benign race conditions that do not violate atomicity.

Bacon *et al* developed Guava [5], an extension to the Java language with a form of monitor capable of sharing object state in a way that prevents race conditions. The Atomizer would work very well for languages like Guava, since language-enforced race freedom would eliminate several common sources of false alarms observed while checking programs written in languages that permit races.

In recent work, Flanagan and Qadeer developed a static type system to verify atomicity in Java programs [25, 24]. In comparison to the Atomizer, the type system provides better coverage and soundness guarantees, but is less expressive (for example, it does not support redundant locking). The type system also requires programmer-inserted annotations that specify properties such as the locking discipline followed by the program.

This type system for atomicity was inspired by the Calvin-R [26] static checking tool for multithreaded programs. Calvin-R supports modular verification of multithreaded programs by annotating each procedure with a specification; this specification is related to the procedure implementation via abstraction relation that combines the notions of simulation and reduction. In ongoing work, the notions of reduction and atomicity are used by Qadeer *et al* [46] to infer concise procedure summaries in an analysis for multithreaded programs.

An alternative approach for verifying atomicity using model-checking is being explored by Hatcliff *et al* [30]. In addition to using Lipton's theory of reduction, they also investigate an approach based on partial order reductions. Their experimental results suggest that the model-checking approach for verifying atomicity is feasible for unit-testing, where the reachable state space is smaller than in integration-testing.

Atomicity is a semantic correctness condition for multithreaded software. In this respect, it is similar to strict serializability [43], a correctness condition for database transactions, and linearizability [31], a correctness condition for concurrent objects. Verifying that an object is linearizable requires full program verification. We hope that our analysis for atomicity can be leveraged to develop lightweight checking tools for related correctness conditions.

Artho *et al* [1] have developed a dynamic analysis tool to identify one class of "higher-level races". The analysis is based on the notion of *view consistency*. Intuitively, a view is the set of variables accessed within a synchronized block. Thread A is view consistent with B if all views from the execution of A, intersected with the maximal view of B, are ordered by subset inclusion. Violations of view consistency can indicate that a program may be using shared variables in a problematic way. View consistency violations can also be detected statically [52]. ESC/Java has been extended to catch a different notion of higher-level races, where a stale value from one synchronized block is used in a subsequent synchronized block [9].

In recent work, Wang and Stoller [53] developed several algorithms for checking atomicity dynamically, including the basic algorithm described in Section 3.3. Their work focuses primarily on more expressive algorithms that can verify additional execution sequences as serializable. They have not yet applied their algorithms to large programs. Our experiments on large programs motivated the development of several crucial improvements to the basic algorithm, such as our support for redundant synchronization operations and write-protected data, and has allowed us to validate the efficiency and effectiveness of our approach.

While our tool checks atomicity, other researchers have proposed using atomicity as a language primitive, essentially implementing the serialized semantics $\mapsto$. Lomet [40] first proposed the use of atomic blocks for synchronization. The Argus [39] and Avalon [17] projects developed language support for implementing atomic objects. Persistent languages [2, 3] are attempting to augment atomicity with data persistence in order to introduce transactions into programming languages. A more recent approach to supporting atomicity uses lightweight transactions implemented in the run-time system [29]. An alternative is to generate synchronization code automatically from high-level specifications [14].

7 Conclusions

Developing reliable multithreaded software is notoriously difficult, because concurrent threads often interact in unexpected and erroneous ways. Programmers try to avoid unintended interactions by designing methods and interfaces that are atomic, but traditional testing techniques are inadequate for verifying atomicity.

This paper presents a dynamic analysis designed to catch atomicity violations would be missed by traditional testing or (static or dynamic) race-detection techniques. This analysis has been implemented and applied to a range of benchmark programs, and has successfully detected atomicity violations in these programs. In addition, our experimental results suggest that over 90% of the methods in our benchmarks are atomic, which validates our hypothesis that atomicity is a fundamental design principle in multithreaded programs.

For future work, we hope to study *hybrid* atomicity checkers based on a synthesis of the dynamic and static approaches. In one combination, a static type-based analysis may verify many expected race-freedom and atomicity properties, and the dynamic atomicity checker could then focus on the unverified residue. For race detection, this hybrid approach has reduced the instrumentation overhead by an order of magnitude [51, 42]; we expect comparable improvements when checking atomicity.

Acknowledgments. We thank Martín Abadi, Shaz Qadeer, Rob O'Callahan, and Scott Stoller for valuable comments on this work. We also thank Christof von Praun for his assistance in collecting test programs. 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.

8 References

[1] C. Artho, K. Havelund, and A. Biere. High-level data races. In *The First International Workshop on Verification and Validation of Enterprise Information Systems*, 2003.

[2] M. P. Atkinson, K. J. Chisholm, and W. P. Cockshott. PS-Algol: an Algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31, 1981.

[3] M. P. Atkinson and D. Morrison. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems*, 7(4):539–559, 1985.

[4] R.-J. Back. A method for refining atomicity in parallel algorithms. In *PARLE 89: Parallel Architectures and Languages Europe*, volume 366 of *Lecture Notes in Computer Science*, pages 199–216. Springer-Verlag, 1989.

[5] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 382–400, 2001.

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

[7] C. Boyapati, R. Lee, and M. Rinard. A type system for preventing data races and deadlocks in Java programs. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 211–230, 2002.

[8] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the ACM Conference on Object-*

Oriented Programming, Systems, Languages and Applications, pages 56–69, 2001.

[9] M. Burrows and K. R. M. Leino. Finding stale-value errors in concurrent programs. Technical Note 2002-004, Compaq Systems Research Center, 2002.

[10] A. T. Chamillard, L. A. Clarke, and G. S. Avrunin. An empirical comparison of static concurrency analysis techniques. Technical Report 96-084, Department of Computer Science, University of Massachusetts at Amherst, 1996.

[11] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridhara. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 258–269, 2002.

[12] E. Cohen and L. Lamport. Reduction in TLA. In *Proceedings of the International Conference on Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 1998.

[13] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, 1996.

[14] X. Deng, M. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *International Conference on Software Engineering*, pages 442–452, 2002.

[15] T. W. Doeppner, Jr. Parallel program correctness through refinement. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 155–169, 1977.

[16] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. Technical Report 94-045, Department of Computer Science, University of Massachusetts at Amherst, 1994.

[17] J. L. Eppinger, L. B. Mummert, and A. Z. Spector. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.

[18] L. Fajstrup, E. Goubault, and M. Raussen. Detecting deadlocks in concurrent systems. In D. Sangiorgi and R. de Simone, editors, *Proceedings of the International Conference on Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 332–347. Springer-Verlag, 1998.

[19] C. Flanagan and M. Abadi. Object types against races. In J. C. M. Baeten and S. Mauw, editors, *Proceedings of the International Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 288–303. Springer-Verlag, 1999.

[20] C. Flanagan and M. Abadi. Types for safe locking. In S. D. Swierstra, editor, *Proceedings of European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108. Springer-Verlag, 1999.

[21] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[22] C. Flanagan and S. N. Freund. Detecting Race Conditions in Large Programs. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 90–96, 2001.

[23] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 234–245, 2002.

[24] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 338–349, 2003.

[25] C. Flanagan and S. Qadeer. Types for atomicity. In *Proceedings of the ACM Workshop on Types in Language Design and Implementation*, pages 1–12, 2003.

[26] S. N. Freund and S. Qadeer. Checking concise specifications for multithreaded software. In *Workshop on Formal Techniques for Java-like Programs*, 2003.

[27] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[28] D. Grossman. Type-safe multithreading in Cyclone. In *Proceedings of the ACM Workshop on Types in Language Design and Implementation*, pages 13–25, 2003.

[29] T. L. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 388–402, 2003.

[30] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation*, 2004.

[31] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[32] C. Hoare. Monitors: an operating systems structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[33] Java Grande Forum. Java Grande benchmark suite. Available from `http://www.javagrande.org/`, 2003.

[34] T. Kistler and J. Marais. WebL – a programming language for the web. In *Proceedings of the International World Wide Web Conference*, volume 30 of *Computer Networks and ISDN Systems*, pages 259–270. Elsevier, 1998.

[35] N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, 1998.

[36] N. Kobayashi, S. Saito, and E. Sumii. An implicitly-typed deadlock-free process calculus. In C. Palamidessi, editor, *Proceedings of the International Conference on Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 489–503. Springer-Verlag, 2000.

[37] L. Lamport and F. B. Schneider. Pretending atomicity. Research Report 44, DEC Systems Research Center, 1989.

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

[39] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proceedings of the Symposium on Operating Systems Principles*, pages 111–122, 1987.

[40] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *Language Design for Reliable Software*, pages 128–137, 1977.

[41] J. Misra. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. Springer-Verlag, 2001.

[42] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 167–178, 2003.

[43] C. Papadimitriou. *The theory of database concurrency control*. Computer Science Press, 1986.

[44] Polyspace technologies, 2003. Available at `http://www.polyspace.com`.

[45] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, pages 179–190, 2003.

[46] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, 2004.

[47] 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.

[48] Standard Performance Evaluation Corporation. SPEC benchmarks. Available from `http://www.spec.org/`, 2003.

[49] N. Sterling. Warlock: A static data race analysis tool. In *Proceedings of the USENIX Winter Technical Conference*, pages 97–106, 1993.

[50] C. von Praun and T. Gross. Object race detection. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 70–82, 2001.

[51] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 115–128, 2003.

[52] C. von Praun and T. Gross. Static detection of atomicity violations in object-oriented programs. In *Workshop on Formal Techniques for Java-like Programs*, 2003.

[53] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *Proceedings of the Workshop on Runtime Verification*, volume 89(2) of *Electronic Notes in Computer Science*. Elsevier, 2003.

[54] World Wide Web Consortium. Jigsaw. Available from `http://www.w3c.org`, 2001.

A Proof of Theorem 2

We start by defining two additional transition relations: $\Sigma \Rightarrow_t \Sigma'$, which performs an instrumented step of thread t, and $\Sigma \models\!\!\Rightarrow \Sigma'$, which is a serialized variant of the instrumented semantics $\Rightarrow$.

Additional relations: $\Sigma \Rightarrow_t \Sigma'$ **and** $\Sigma \models\!\!\Rightarrow \Sigma'$

[INS STEP T]

$$\frac{T(t, \Pi(t), a, \pi') \quad \sigma \rightarrow_t^a \sigma' \quad (\sigma, \phi, \Pi) \Rightarrow_t^a \phi'}{(\sigma, \phi, \Pi) \Rightarrow_t (\sigma', \phi', \Pi[t := \pi'])}$$

[INS SERIAL STEP]

$$\frac{(\sigma, \phi, \Pi) \Rightarrow_t (\sigma', \phi', \Pi') \quad \forall u \neq t.\ A(\Pi(u)) = 0}{(\sigma, \phi, \Pi) \models\!\!\Rightarrow (\sigma', \phi', \Pi')}$$

We introduce three state predicates $N(t)$, $R(t)$, and $L(t)$, where $N(t)$ means that thread t is not in an atomic block, and $R(t)$ and $L(t)$ mean that thread t is in the right-mover and left-mover parts of an atomic block, respectively. The following Reduction Theorem formalizes five conditions that are sufficient to conclude that all atomic blocks are reducible.

The statement of this theorem uses some additional notation. For two actions $b, c \subseteq State \times State$, we say that b *right-commutes* with c if for all $\Sigma_1, \Sigma_2, \Sigma_3$, whenever $(\Sigma_1, \Sigma_2) \in b$ and $(\Sigma_2, \Sigma_3) \in c$, then there exists Σ_2' such that $(\Sigma_1, \Sigma_2') \in c$ and $(\Sigma_2', \Sigma_3) \in b$. The action b *left-commutes* with the action c if c right-commutes with b. We also define the *left restriction* $\rho \cdot b$ and the *right restriction* $b \cdot \rho$ of an action b with respect to a set of states $\rho \subseteq State$.

$$\begin{aligned} \rho \cdot b &\stackrel{\text{def}}{=} \{(\Sigma, \Sigma') \in b \mid \Sigma \in \rho\} \\ b \cdot \rho &\stackrel{\text{def}}{=} \{(\Sigma, \Sigma') \in b \mid \Sigma' \in \rho\} \end{aligned}$$

THEOREM 5 (REDUCTION). *Suppose that for all $t, u \in Tid$ with $t \neq u$:*

A1. *$R(t)$, $L(t)$, and $N(t)$ form a partition of $State$.*

A2. *$(L(t) \cdot \Rightarrow_t \cdot R(t))$ is empty.*

A3. *$(\Rightarrow_t \cdot R(t))$ right-commutes with $\Rightarrow_u$.*

A4. *$(L(t) \cdot \Rightarrow_t)$ left-commutes with $\Rightarrow_u$.*

A5. *if $\Sigma \Rightarrow_t \Sigma'$, then $\Sigma \in R(u) \Leftrightarrow \Sigma' \in R(u)$, and $\Sigma \in L(u) \Leftrightarrow \Sigma' \in L(u)$.*

Suppose further that $\Sigma_0 \Rightarrow^* \Sigma$ and Σ_0 and Σ are in $N(t)$ for all $t \in Tid$. Then $\Sigma_0 \models\!\!\Rightarrow^* \Sigma$.

PROOF: See [25].

We now leverage this theorem to show that every instrumented execution trace is reducible.

RESTATEMENT OF THEOREM 2 (INSTRUMENTED REDUCTION) *If $(\sigma_0, \phi_0, \Pi_0) \Rightarrow^* (\sigma, \phi, \Pi)$ and $\neg\mathcal{A}(\Pi)$ then $(\sigma_0, \Pi_0) \mapsto^* (\sigma, \Pi)$.*

PROOF: We define the predicates $N(t)$, $R(t)$, and $L(t)$ necessary to apply Theorem 5 (Reduction) as follows:

$$\begin{aligned} N(t) &\stackrel{\text{def}}{=} \{(\sigma, \phi, \Pi) \mid A(\Pi(t)) = 0\} \\ R(t) &\stackrel{\text{def}}{=} \{(\sigma, \phi, \Pi) \mid A(\Pi(t)) \neq 0 \wedge \phi(t) = \mathit{InRight}\} \\ L(t) &\stackrel{\text{def}}{=} \{(\sigma, \phi, \Pi) \mid A(\Pi(t)) \neq 0 \wedge \phi(t) = \mathit{InLeft}\} \end{aligned}$$

These predicates satisfy the following five requirements of Theorem 5, for $t, u \in Tid$ with $t \neq u$:

A1. They clearly partition $State$.

A2. $(L(t) \cdot \Rightarrow_t \cdot R(t))$ is empty, since $\phi(t)$ is never set to $\mathit{InRight}$ while within an atomic block.

A3. $(\Rightarrow_t \cdot R(t))$ right-commutes with $\Rightarrow_u$, since if $R(t)$ holds after an action of thread t, then that action must be by one of the rules [INS ACCESS PROT], [INS ACQUIRE], [INS RE-ENTER], [INS ENTER], or [INS OTHER], all of which right-commute with $\Rightarrow_u$.

A4. $(L(t) \cdot \Rightarrow_t)$ left-commutes with $\Rightarrow_u$, since if $L(t)$ holds before an action of thread t, then that action must be by one of the rules [INS ACCESS PROT], [INS RELEASE], [INS RE-ENTER], or [INS OTHER], all of which left-commute with $\Rightarrow_u$.

A5. if $\Sigma \Rightarrow_t \Sigma'$, then $\Sigma \in R(u) \Leftrightarrow \Sigma' \in R(u)$ and $\Sigma \in L(u) \Leftrightarrow \Sigma' \in L(u)$, since a step by thread t does not change $\phi(u)$ or $\Pi(u)$.

Hence by Theorem 5 (Reduction), $(\sigma_0, \phi_0, \Pi_0) \models\!\!\Rightarrow^* (\sigma, \phi, \Pi)$, and therefore $(\sigma_0, \Pi_0) \mapsto^* (\sigma, \Pi)$.