Abstract

Multithreaded programs can have subtle errors that result from undesired interleavings of concurrent threads. A common technique programmers use to prevent these errors is to ensure that certain blocks of code are atomic. A block of code is atomic if every execution is equivalent to a serial execution in which no other thread’s instructions are interleaved with the code. Atomic blocks of code are amenable to sequential reasoning and are therefore significantly simpler to analyze, verify, and maintain.

This paper presents a system for automatically detecting atomicity violations in Java programs without requiring any specifications. Our system infers which blocks of code must be atomic and detects violations of atomicity of those blocks. The paper first describes a synchronization pattern in programs that is likely to indicate a violation of atomicity. The paper then presents a static analysis for detecting occurrences of this pattern.

We tested our system on over half a million lines of popular open source programs, and categorized the resulting atomicity warnings. Our experience demonstrates that our system is effective. It successfully detects all the previously known atomicity errors in those programs as well as several previously unknown atomicity errors. Our tool also detects badly written code whose atomicity depends on assumptions that might not hold in future versions of the code.

A key underlying idea behind our system is that we have identified a pattern of synchronization in Java code that is highly correlated with atomicity violations and that can be detected efficiently using a static analysis. The pattern suggests that a block is intended to be atomic and yet may have non-serializable interleavings. Our tool uses an efficient interprocedural static analysis to detect this pattern. It then reports instances of the pattern as potential violations of atomicity.

Much previous work on detecting and preventing synchronization errors has focused on data races. A data race occurs when two threads concurrently access the same shared data without synchronization, and at least one of the accesses is a write. Unfortunately, absence of data races does not ensure absence of synchronization errors because a program without data races can still have atomicity violations [5].

A common technique programmers use to prevent synchronization errors is that they ensure that certain blocks of code are atomic. A block of code is atomic if every execution is equivalent to a serial execution in which no other thread’s instructions are interleaved with the code. Atomic blocks of code are amenable to sequential reasoning and are much simpler to analyze, verify, and maintain because one does not have to consider an enormous number of interleavings.

This paper presents a tool for automatically detecting atomicity violations in existing Java programs using a static analysis. Most existing Java programs do not contain specifications that declare which blocks of code must be atomic. Our tool automatically identifies blocks of code that the programmers intended to be atomic and uses a static analysis to detect violations of the intended atomicity. In addition, even when there is no error, our tool detects badly written code whose atomicity depends on assumptions that might not hold in future versions of the code, and which might thus lead to bugs in future versions of the code.

Abstract

Categories and Subject Descriptors D.1.3 [Programming Techniques]: Concurrent Programming; D.2.4 [Software Engineering]: Software/Program Verification; D.2.5 [Software Engineering]: Testing and Debugging; D.3.1 [Programming Languages]: Formal Definitions and Theory; D.3.4 [Programming Languages]: Processors; F.3.1 [Logics and Meanings of Programs]: Specifying, Verifying, and Reasoning about Programs

General Terms Algorithms, Design, Experimentation, Reliability, Verification

Keywords

1. Introduction

Multithreaded programming is becoming increasingly common because of the advances in parallel hardware technology. But multithreaded programming is difficult and error prone. Multithreaded programs synchronize operations on shared mutable data to ensure that the operations execute atomically. Failure to correctly synchronize such operations can lead to errors. Synchronization errors in multithreaded programs are timing-dependent non-deterministic bugs that are among the most difficult programming errors to detect, reproduce, and eliminate.
This paper identifies a synchronization pattern in Java code that might not hold in future versions of the code. The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.

The false positive rate is small and is under 50%.
This pattern is likely to contain an atomicity violation. The context lock A indicates that the entire block of code where lock A is held must be atomic. Recall that lock acquires and releases in Java are block structured. However, the multiple acquisitions and releases of the inner lock B are likely to violate that atomicity. In particular, during the time that the lock on B is released and then reacquired, the data protected by the lock B could potentially be modified by another thread and therefore their previously read values could become stale.

Our experience does indicate that the occurrence of this pattern usually implies an atomicity violation. Furthermore, even when there is no error, we hold that in most cases the pattern indicates bad programming style. If the block is intended to be atomic, it must acquire all its locks first before it releases any of its locks. If the block is not intended to be atomic, it must not hold any locks for the duration of the block. Thus, even if there is no error, the atomicity of the code depends on assumptions that might not hold in future versions of the code.

### 2.3 Pattern Variant

The pattern in Figure 3 can be relaxed to catch more errors, possibly at the expense of generating more false positives. The pattern variant occurs when a thread, while holding a lock A, acquires and releases a lock B1 and then acquires and releases a lock B2. A must be different from both B1 and B2. B1 and B2 may be the same lock or different locks. Note that if B1 and B2 are the same lock, then this is identical to the original pattern described in Section 2.2.

This relaxed variant also indicates bugs because the atomic block might rely on a concurrent snapshot of data protected by B1 and B2. The atomicity may be violated because of the same reasons as above. Furthermore, even if there is no error, this indicates bad programming because of the same reasons as above.

In addition to detecting occurrences of the original pattern, our tool also detects occurrences of the relaxed pattern variant when a flag is switched on.

---

**Figure 4.** Example call graph with five methods. Method m1 calls method m2 while holding the lock on A. Our analysis shows that the lock on B is acquired and released twice while method m1 holds the lock on A.

---

### 3. Approach

Given the pattern described in Section 2, we developed a tool that quickly and automatically detects instances of this pattern in Java code.

#### 3.1 Locking Within a Method

Consider the code in Figure 1, which contains an atomicity violation. Our tool detects that the method acquires and releases the lock on point in Line 4 and again in Line 5. We use a flow-sensitive analysis because we only want to detect instances where control flow passes through both acquisitions of the lock on point. For example, the following code clearly does not acquire and release the lock twice.

```java
if (b)
    r1 = point.distanceTo(start);
else
    r2 = point.distanceTo(end);
```

Our dataflow analysis tracks information about which locks are acquired and released twice (a). After propagating the results up the call graph, we see that the lock on B is acquired and released twice while the lock on A is held in method m1.

---

**Figure 5.** Results computed while analyzing the call graph of Figure 4. First we build the set of locks acquired and released by each method (a). Then we use a static dataflow analysis to find which locks are acquired and released twice (b). After propagating the results up the call graph, we see that the lock on B is acquired and released twice while the lock on A is held in method m1.
for (int i = 0; i < 2; i++) {
    r[i]=point[i].distanceTo(endpoints[i]);
}

Although both synchronizations occur at the same place in the code, the locks being synchronized are distinct. Since point[1] represents two different locks, it would be incorrect to mark it as having been acquired and released twice. Our analysis solves this problem efficiently by noticing that the variable i is modified, and unmarking point[i]. (If point[i] is already marked twice then we leave it marked, since it was already marked twice before i was modified.)

In general, our analysis unmarks a lock variable whenever its value changes, or whenever a dependent value changes (such as an array index). Thus we can use an efficient syntax-based alias detection as long as we unmark any lock that depends on a variable or field that changes values. In practice, this technique is sufficient to avoid most if not all false positives that arise from imprecise alias analysis.

At the end of the method contains in Figure 1, our dataflow analysis has point marked twice. Since contains synchronizes this, we have discovered an instance of the pattern. While the lock on this is held, the lock on point is acquired and released twice.

### 3.2 Locking Across Methods

The above analysis considers analyzing only a single method. Our full analysis also detects the pattern when it occurs across multiple methods. For example, consider the call graph of Figure 4. In this call graph, method m1 holds the lock on A and calls method m2, which calls methods m3 and m4 in sequence. Methods m3 and m4 each hold the lock on A while calling m5, which acquires and releases the lock on B. The locking is all structured, so each method releases its lock before returning.

First, our analysis determines which locks each method may eventually acquire and release. For example, the methods m3 and m4 each acquire and release the lock on B by calling m5. Thus, m2 acquires and releases the lock on B as well, and the analysis continues up the call graph in this fashion. Figure 5(a) presents the initial locks that are known to be acquired by each method, along with the results of propagating this information up the call graph.

Next our analysis applies the dataflow analysis described above to each method to find out which locks are acquired and released twice. This analysis depends on the results in the second column of Figure 5(a) for determining which locks are acquired and released during method calls. For example, the dataflow analysis of method m2 uses the fact that each call to m3 and m4 acquires and releases the locks on both A and B. Thus, our analysis determines that m2 acquires and releases A and B twice each. The first column of Figure 5(b) tabulates the results of the dataflow analysis. Note that method m2 does not exhibit the pattern because neither lock is held while the other is acquired and released twice.

Finally, our analysis discovers whether either of the locks are acquired and released in the context of a different lock, and thus exhibits the pattern. To do this, it propagates the information about locks that are acquired and released twice. For example, m2 acquires and releases A and B twice, so any method that calls m2 also acquires and releases A and B twice. However, m1 already holds the lock on A when it calls m2, so the reentrant locking on A is ignored. Therefore only B is propagated to m1, as the second column of Figure 5(b) shows. Method m1 holds the lock on A while locking and releasing B twice, which is an instance of the pattern described in Section 2. Our analysis detects the pattern, and it reports the lines of code where A and B are locked.

### 3.3 Call Graph Propagation

The above analysis depends on propagating locks up the call graph. Our analysis represents locks according to their syntax, which can vary between methods. For example, in Figure 1 the method distanceTo synchronizes the lock on this, which corresponds to the lock on point in method contains. When propagating locks between methods, our analysis converts formal parameter names (including this) into the actual parameters at the call site. We replace names that can’t be converted (such as local variable names) with a generic unknown lock.

Consider the code in Figure 1, and suppose that a method calls contains while holding the lock on point. Then by propagating the lock information in contains back to this method, it appears that there is no atomicity violation at all since the reentrant locking of point is ignored. However, the method contains is intended to be atomic. It will eventually be used with a method that does not hold the lock on point. Therefore our analysis detects and reports the atomicity violation of contains, even if the method that calls it holds the lock on point.

### 4. Static Analysis Formalism

This section formally describes the static analysis we use to detect the synchronization pattern that indicates an atomicity violation. First we describe an intraprocedural version of the analysis. We then extend this to an interprocedural analysis and describe how we use the resulting information to generate atomicity warnings.

#### 4.1 Preliminaries

In Java, objects are associated with locks. The Java synchronized keyword defines a block of code in which a given object’s lock is held. The lock is acquired at the beginning of the block (possibly causing the thread to wait) and released at the end of the block. Similarly, an entire method may be synchronized, which implicitly puts the entire method body into a block synchronized on the this object. We assume that all such implicit synchronization is transformed into explicit synchronization blocks. We define a locking expression (LE) according to the following grammar

\[
LE ::= v | \text{this} | LE.f | LE[c]
\]

where \(v\) is a program variable, \(f\) is a field, and \(c\) is an integer expression. Locking expressions can be arbitrarily long \((v.f_1.f_2.f_3...\), so we define Locks as the set of all such locking expressions up to a given size bound. If a locking expression exceeds the bound, we denote it with the special lock *, which we take to be in the set Locks. We model method calls as a statement of the form CALL \(e.m(...)\) where \(e\) is an expression and \(m\) is a method.

We model synchronization as a statement of the form SYNC \(e \# b\) where \(e\) is a locking expression and \#b is a synchronization block. The statement SYNC \(\# b\) refers to a synchronization block \#b that does not synchronize on a locking expression within the bounds defined above.

We assume that a preceding compiler stage has constructed a call graph (we use Class Hierarchy Analysis [3]) and control flow graphs for the program. The program points immediately before and after a statement \(st\) are respectively denoted \(\text{pred}(st)\) and \(\text{post}(st)\). The set of all predecessors of a statement \(st\) is denoted \(\text{pred}(st)\).

Let entry\_m denote the set of method and synchronization block entry statements in method \(m\). (Note that the inclusion of synchronization block entry points is different from the usual definition of entry\_m.) Let \(A(p)\) denote the dataflow facts at a program point \(p\). (The dataflow facts are defined in the next subsection.) Let \(A(m)\)
The dataflow facts denote the flow-sensitive forward dataflow analysis that tracks some facts. Our intraprocedural analysis detects the synchronization pattern of a block synchronizing a different lock, then the lock enters \( P \) the first time they are acquired and released. After the second time, they enter \( T \). If a lock enters \( R \) and \( T \) in the scope of a block synchronizing a different lock, then the lock enters \( P \), which indicates that the pattern has been detected.

The dataflow facts \( \langle R, T, P \rangle \) form a lattice where \( \langle R_1, T_1, P_1 \rangle \subseteq \langle R_2, T_2, P_2 \rangle \) if and only if \( R_1 \subseteq R_2 \land T_1 \subseteq T_2 \land P_1 \subseteq P_2 \).

The join operator is defined such that
\[
\langle R_1, T_1, P_1 \rangle \lor \langle R_2, T_2, P_2 \rangle = \langle R_1 \cup R_2, T_1 \cup T_2, P_1 \cup P_2 \rangle.
\]

The dataflow equations are presented below:

\[
A(\bullet s t) = \begin{cases} 
(\emptyset, \emptyset, \emptyset) & \text{if } st \in \text{entry}_m \\
\bigcup_{st^* \in \text{pred}(st)} A(st^* \bullet) & \text{otherwise}
\end{cases}
\]

These equations set the initial dataflow facts \( \langle R, T, P \rangle \) to empty sets for every method entry and every synchronization block entry point. (Recall that \( \text{entry}_m \) is defined as the set of all entry points to the method \( m \) and to synchronization blocks in \( m \).) Every other program point just before a statement is computed as the join of the dataflow facts just after all predecessors of that program point.

The transition functions (defined below) are used to compute the dataflow facts just after a given statement.

Note that these dataflow equations differ from traditional dataflow equations by defining the initial dataflow facts of synchronization blocks. We do this in order to determine which locks in the set \( T \) to add to the set \( P \). For a lock to enter \( P \), it must have entered \( R \).
and \( T \) while inside a synchronization block of a different lock. By setting the initial facts of all synchronization blocks to the empty set, we know that all locks in \( T \) must have entered \( R \) and \( T \) while in the current synchronization block. This enables us to determine which locks should enter \( P \).

Figure 6(a) presents the transfer functions. The transfer functions for the \( \text{SYNC} \) statements look up the end of the dataflow analysis at the end of the synchronization block \( \text{sib} \), denoted \( (R_{\text{sib}}, T_{\text{sib}}, P_{\text{sib}}) \). For a synchronization statement \( \text{SYNC} e \ \text{abc} \), the set of previously released locks \( R \) is modified to include all locks released in \( \text{sib} \), plus \( e \). The set of potential witnesses \( T \) is updated to include any of these locks that were already present in \( R \). The set of actual witnesses \( P \) is updated to include locks that witness the pattern in \( \text{sib} \) as well as potential witnesses in \( \text{sib} \) that now witness the pattern with \( e \) as context. However, \( e \) itself is not added to \( P \) because the pattern requires that witnesses be distinct from their context. The statement \( \text{SYNC} * \ \text{abc} \) is treated similarly, with * assumed to be a lock that is distinct from every other lock, including other instances of *. Thus * may be used as a context but never a witness of the pattern.

The transfer function for an assignment \( x := y \) removes from \( R \) all locks that depend on \( x \). A lock \( e \) depends on \( x \) if it satisfies \( \text{DEPENDS}(e, x) \), which is defined at the the top of Figure 7. \( \text{DEPENDS}(e, x) \) holds whenever \( e \) is composed of \( x \) in some way, as either a field or an array access of \( x \), or if \( x \) is used to index an array in \( e \).

### 4.3 First Interprocedural Extension

The intraprocedural technique described above is unable to detect patterns where locks are acquired in different methods. For example, the code in Figure 1 will result in the set \( P \) always being empty, despite the presence of the pattern. To remedy this, we introduce an interprocedural analysis. For each method \( m \), the analysis generates \( \text{LOCKSHIELD}(m) \), the set of locks that may be acquired by \( m \) or methods that \( m \) calls. This additional information can then be used during the intraprocedural analysis at call sites.

One important caveat is that locking expressions can refer to method-local variables, including \( \text{this} \) and formal parameters. Before using the value of \( \text{LOCKSHIELD} \) from another method, each lock needs to be translated to the current context. For example, the method \( \text{distanceTo} \) in Figure 1 locks \( \text{this} \). In the method \( \text{contains} \), that lock should be translated to the target of the call. \( \text{point} \). Figure 7 defines the function \( \text{TRANSLATELOCK}(x) \), which translates locking expression \( x \) for the call site \( \text{at} \). \( \text{TRANSLATELOCK} \) translates field and array accesses recursively. It translates \( \text{this} \) to the target of the method call. It translates formal parameters to the arguments of the method call. An expression containing any other method-local variable is translated to *. Additionally, any translation that would result in a locking expression that exceeds the size bound results in *.

Figure 7 presents the equation for computing \( \text{LOCKSHIELD} \). For each method \( m \), \( \text{LOCKSHIELD}(m) \) contains all locking expressions acquired in \( \text{SYNC} \) statements in \( m \), as well as the translations of all of its callees' locks. We define \( \text{LOCKSHIELD} \) to be the least solution to this equation, which can be efficiently computed with a standard fixed point algorithm.

The new transfer function for the interprocedural analysis is shown in Figure 6(b). The set \( L \) is defined as the set of translated locks from all of the callees. The new set of released locks \( R \) therefore includes \( L \), and the new set of potential witnesses \( T \) includes every lock in \( L \) that was already in \( R \). The set of locks witnessing the pattern \( P \) is left unmodified.

### 4.4 Second Interprocedural Extension

The interprocedural extension described above accounts for locks acquired by callees, but it does not immediately detect the pattern when a lock becomes a potential witness in a callee. For example, suppose method \( m \) acquires a lock \( x \) and calls a method \( m' \), which acquires and releases a lock \( y \) twice. The pattern is exhibited with \( x \) as context and \( y \) as witness. The interprocedural extension above will terminate with the final state of \( m \) as \( \{x, y\} \), \( \emptyset, \emptyset \) and the final state of \( m' \) as \( \{\{y\}, \{y\}, \emptyset\} \). Both methods have an empty set of pattern witnesses, but the lock acquired in \( m \) serves as context for the pattern witnessed in \( m' \). A final analysis propagates the set \( T \) of potential witnesses back up the call graph in search of a context.

The final analysis operates on an expanded call graph, where each method and synchronization block preserves a unique node. Edges are either method calls or control flow paths to embedded synchronization blocks. Thus the path from a method to a callee might pass through a number of synchronization block nodes. Here we extend the definition of \( \text{CALLEES}(n) \) for the case where \( n \) is a synchronization block. Let \( \text{SYNC}(n) \) denote the set of synchronization block nodes that are successors of node \( n \) in the expanded call graph. For each node in the expanded call graph, the final analysis tracks a pair of sets, \( \{T, P\} \), where \( T \) and \( P \) are as usual the sets of potential and actual witnesses, respectively. For a synchronization node \( n \), define \( A(n) \) as \( \emptyset, \emptyset, \emptyset \).

Figure 8 presents the equations for computing \( \text{FINAL}_P \) and \( \text{FINAL}_T \) for each node of the expanded call graph. \( \text{FINAL}_T \) computes the set of potential witnesses for each node. This is simply propagated from the callees of the node. Synchronization blocks either confirm or eliminate a potential witness, so the only source of potential witnesses is from callees. \( \text{FINAL}_P \) computes the set of confirmed witnesses for each node. These are either potential witnesses that are confirmed by a synchronization block, or previously confirmed witnesses that pass through a synchronization block. Witnesses passing through a synchronization block are eliminated if they are the same as the lock being synchronized by \( s \). Java’s reentrant locking would render such witnesses as no-ops.

Note that witnesses in \( \text{FINAL}_P \) are never propagated between methods, even though they might be eliminated by synchronization blocks further up the call hierarchy. Many methods are designed to be atomic in any calling context, and the warnings our tool gen-

\[
\text{FINAL}_P(n) = A(n) \cup \left( \bigcup_{x \in \text{SYNC}(n)} \{\text{FINAL}_T(x) \cup \text{FINAL}_P(x)\} \setminus \{e\} \right)
\]

\[
\text{FINAL}_T(n) = A(n) \cup \left( \bigcup_{m \in \text{CALLEES}(n)} \{\text{TRANSLATELOCK}(x) \mid x \in \text{FINAL}_T(m)\} \setminus \{\ast\} \right)
\]
<table>
<thead>
<tr>
<th>st</th>
<th>([st](\langle R,T,P \rangle))</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{SYNC} e sb</td>
<td>((R \cup R_{sb} \cup {e}, T \cup (R \times (R_{sb} \cup {e})), P \cup (P_{sb} \cup T_{sb}) \setminus \text{FILL}(e)))</td>
</tr>
<tr>
<td>\text{SYNC} + sb</td>
<td>((R \cup R_{sb}, T \cup (R \times R_{sb}), P \cup P_{sb} \cup T_{sb}))</td>
</tr>
<tr>
<td>CALL (c.m'(p))</td>
<td>((R \cup L, T \cup (R \times L), P))</td>
</tr>
<tr>
<td>all others</td>
<td>((R, T, P)) for (A(sb) = \langle R_{sb}, T_{sb}, P_{sb} \rangle) and (L = \bigcup_{m'' \in \text{CALLES}(m'')} {\text{TRANSLATELOCK}_{m''}(x) \mid x \in \text{LOCKSHELD}(m'')})</td>
</tr>
</tbody>
</table>

**Figure 9.** Transfer functions for detecting the pattern variant

\[
\text{VARFINAL}_P(n) = A(n) \cup \bigcup_{s \in \text{STYLE} \cdot sb \in \text{SYNC}(n)} (\text{VARFINAL}_T(s) \cup \text{VARFINAL}_P(s)) \setminus \text{FILL}(e)
\]

\[
\text{VARFINAL}_T(n) = A(n) \cup \bigcup_{m \in \text{CALLES}(n)} \{\text{TRANSLATELOCK}(x), \text{TRANSLATELOCK}(y) \mid \langle x, y \rangle \in \text{VARFINAL}_T(m) \setminus \text{FILL}(+)\}
\]

\[
\text{FILL}(x) = \text{Locks} \times \{x\} \cup \{x\} \times \text{Locks}
\]

**Figure 10.** Equations for computing final sets of witnesses for the pattern variant, along with the helper function \(\text{FILL}\).

ertes can assist in the development of these methods. Therefore, once a witness reaches a method boundary it is considered an actual instance of the pattern.

### 4.5 Atomicity Warning Generation

After the above analyses, \(\text{FINAL}_P(m)\) is the set of pattern witnesses of each method \(m\). The final set of witnesses is then

\[
\bigcup_{m \in \text{METHODS}} \text{FINAL}_P(m).
\]

The witness may not have been locked in \(m\), so the same witness could appear in \(\text{FINAL}_P\) for multiple methods. We report only one atomicity warning for each location in the code where the witness was locked for the second time. Each warning contains the locations of both witness lock locations as well as the context.

### 4.6 Detecting the Pattern Variant

Section 2.3 describes a variant on the pattern where the witness synchronizations are allowed to operate on two different locks. The above analysis can be modified to check for this variant. Each witness and potential witness is now a pair of locks \(\langle a, b \rangle \in \text{Locks} \times \text{Locks}\), where \(a\) was locked and released before \(b\) was locked. Thus the sets \(T\) and \(P\) are subsets of \(\text{Locks} \times \text{Locks}\). The subset relation and join operation remain the same. Figure 9 presents the transfer functions for the modified analysis and Figure 10 presents the final analysis equations.

The variant pattern analysis has the potential to produce more atomicity warnings, since many more combinations of locks are possible. Like above, we limit the output to one warning per location of the second witness lock.

### 5. Experience

This section presents our experience using our analysis tool on a variety of Java programs. We ran our experiments on an Ubuntu 10.04 machine with a Pentium 4 3.4 GHz processor and 1 GB memory using IcedTea Java 1.8.

#### 5.1 Methodology

We implemented the analysis of Section 4 as an extension to the Polyglot [15] compiler framework. Our tool combines warnings that involve the same witness lock into a single warning. We ran our tool on a set of Java programs:

- **elevator**: A concurrent elevator simulation [21].
- **tsp**: A parallel solution to the traveling salesman problem [21].
- **sor**: A parallel scientific computation application [21].
- **ObjectDraw**: A concurrent object drawing library used for teaching Java [2].
- **Risk**: A computer board game [17].
- **Arbaro**: A utility for creating and rendering realistic trees [1].
- **TVSchedulerPro**: A program for capturing data from a digital tuner device according to a schedule [19].
- **FreeMind**: A mind mapping application [7].
- **TuxGuitar**: A guitar tablature editor [18].
- **Jose**: A graphical tool for the game chess [11].
- **JFreeChart**: A chart library for Java [10].

Risk, Arbaro, TVSchedulerPro, FreeMind, TuxGuitar, Jose, and JFreeChart are all popular projects at sourceforge.net. Our tool currently supports only Java 1.4, so we used a program to automatically remove the extra features of Java 5 from the TVSchedulerPro source code.

#### 5.2 Classification of Warnings

We ran our tool on the programs and benchmarks above. We manually inspected each warning output by our tool, and categorized it as either a program error or a false positive. We divided the program errors into actual atomicity violations and stylistic problems that are potential bugs as the program develops. We divided the false positives into instances of the pattern that don’t indicate an error, and cases where the pattern is not actually exhibited.
stances of the pattern. In our experience these false positives come
Sometimes our tool produces warnings that do not correspond to in-
Non-Pattern
indicates a problem.
Expected to cause problems now or in the future.
from some programmer attention, but they are benign and are not
have intended a given block to be atomic, or they might be using
will be no error in the code and no indication that there may be
an error in the future. For example, the programmer may not
be an error in the future. For example, the programmer may not
violation. Atomicity violations can cause dramatic failures such
as uncaught exceptions, or more subtle problems such as methods
returning incorrect values. Many atomicity violations are the result
of a method not acquiring a lock on its parameter at the right
granularity, such as the example from Section 2. In all instances of
atomicity violations that we observed, the violation can be resolved
by acquiring an additional lock for the duration of the atomic block.

Stylistic Problem
In some cases the code style suggests a confusion, which may lead
to atomicity violations in the future. In these cases, a suspect syn-
chronization pattern is used, although it may not lead to a violation
of atomicity. For example, an object might not need correct syn-
chronization because it is protected by a lock on a different object.
However, the presence of incorrect synchronization on the encaps-
ulated object suggests that it may rely on that synchronization in a
future version of the code, in which case there will be an atomicity
violation.

There are some standard synchronized classes in the Java such as
Vector and StringBuffer that are often used in an encapsulated
or thread-local setting. Our tool detects patterns involving library
classes such as Vector and StringBuffer and filters them from
the results.

Benign Pattern
Sometimes our tool will correctly identify the pattern but there
will be no error in the code and no indication that there may be
an error in the future. For example, the programmer may not
have intended a given block to be atomic, or they might be using
a nonstandard locking idiom. These false positives might benefit
from some programmer attention, but they are benign and are not
expected to cause problems now or in the future.

Less than 35% of the instances of the pattern identified by our tool
are benign. This supports our hypothesis that the pattern generally
indicates a problem.

Non-Pattern
Sometimes our tool produces warnings that do not correspond to in-
tances of the pattern. In our experience these false positives come
from two sources of imprecision: path sensitivity and conservative
call graphs. Our tool uses a path insensitive technique, so it might
generate warnings that correspond to impossible paths. Addition-
ally, our tool conservatively approximates the call graphs and so it
might consider calls that can never happen. Despite these sources
of imprecision, our tool incorrectly reported a non-pattern less than
15% of the time.

Although our alias analysis is very fast and imprecise, lock vari-
ables are generally used in a simple and straightforward manner in
synchronized code. We did not encounter any false positives as a
result of imprecise alias detection.

5.3 Analysis
The experimental results are tabulated in Figure 11. These results
confirm that the pattern we detect is likely to indicate problems
in the program. Of the 27 warnings that were reported, 9 (33%) were
methods that were intended to be atomic but do not ensure
atomicity through synchronization. There were 6 warnings (22%)
with questionable style, which could lead to problems in future
versions of the code. False positives accounted for under 50% of
the total warnings generated. As expected, the analysis is efficient,
processing over half a million lines of code in a little over three
minutes.

We also built the analysis of the pattern variant as described in Sec-
section 4.6 into our tool and ran it on all benchmarks. The variant anal-
ysis produced a superset of the warnings of the standard analysis.
Figure 12 shows the number of additional warnings generated by
the variant analysis. The additional warnings do not immediately
lead to atomicity violations, but they do indicate code blocks with
complex synchronization patterns that might bear further investiga-
tion or rethinking. We feel that the variant pattern is a useful option
for those wanting more warnings at the expense of a potentially
greater false positive rate.

The warnings tabulated in Figure 11 are described in detail below.

sor
Our tool reported nine warnings for the sor benchmark. We deter-
dined that three of these warnings represent actual atomicity viola-
tions. Two atomicity violations are detected in the addAll methods
de a data structure. These methods take a Collection as a param-
eter but they do not synchronize the parameter. Thus, changes to
the Collection during execution of the addAll methods violates

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines of Code</th>
<th>Total Warnings</th>
<th>Errors and Potential Errors</th>
<th>False Positives</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>elevator</td>
<td>523</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td>tsp</td>
<td>706</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.6</td>
</tr>
<tr>
<td>sor</td>
<td>17690</td>
<td>9</td>
<td>3</td>
<td>6</td>
<td>5.6</td>
</tr>
<tr>
<td>ObjectDraw</td>
<td>5637</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>Risk</td>
<td>10735</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.9</td>
</tr>
<tr>
<td>Arbaro</td>
<td>13760</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>6.6</td>
</tr>
<tr>
<td>TVSchedulerPro</td>
<td>30857</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>16.2</td>
</tr>
<tr>
<td>FreeMind</td>
<td>65161</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>23.1</td>
</tr>
<tr>
<td>TuxGuitar</td>
<td>96035</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>40.3</td>
</tr>
<tr>
<td>Jose</td>
<td>145993</td>
<td>5</td>
<td>0</td>
<td>4</td>
<td>48.8</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>217354</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>40.9</td>
</tr>
<tr>
<td>TOTAL</td>
<td>604451</td>
<td>27</td>
<td>9</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 11. Experimental results of running our analysis on several Java programs. We categorize the warnings into atomicity violations, stylistic problems, benign patterns, and misidentified patterns. Our analysis detected atomicity violations in 33% of the cases. About 85% of the warnings correspond to actual instances of the pattern, and about 65% of these correspond to program errors or potential errors.
atomicity and can lead to exceptions. The third atomicity violation is in a Heap data structure. The insert method is synchronized, but it does not synchronize its parameter.

The remaining six warnings all correctly identify the pattern, but they do not correspond to actual program errors. Four of the warnings are for blocks that are not intended to be atomic but instead explicitly detect certain classes of concurrent modifications. These warnings are reported in sections of the code that relate to load sharing. Finally, there were two warnings that arose from the use of custom lock objects. In these cases, the programmers define their own locking outside of the built-in Java synchronized blocks.

**ObjectDraw**

Our tool identified two atomicity bugs in the ObjectDraw library.

One atomicity bug is in a method that tests whether a line contains a point. While a lock on the line is held, the lock on the point is acquired and released multiple times. The point could be modified by a concurrent thread between acquisitions, making the method return an incorrect result.

The other atomicity bug is in a method that determines whether two bounding boxes intersect. The method acquires a lock on one bounding box but not on the other. Subsequent method calls acquire and release the lock on the other bounding box multiple times. If that bounding box is modified, the method uses stale data and may return an incorrect result.

In addition to these atomicity bugs, our tool detects one stylistic problem, which results from the synchronization of an encapsulated bounding box.

We ran our experiments on a version of ObjectDraw from 2001. The maintainer of the ObjectDraw library has confirmed that the atomicity violations revealed by our tool are previously unknown errors that still exist in the current version of the code.

**Risk**

The Risk program can act as a client when connecting to a game server. If it is kicked from a server, some cleanup is required. During cleanup, a lock is acquired and released twice. This lock protects an object representing the current state of a game. It is locked while closing a battle within a game, and then released and locked again while closing the game itself. These operations are clearly intended to be atomic, such that a new battle is not started after the old one is closed but before the game itself is closed. It is possible for the property to be concurrently set after it is checked, so we conservatively classify this warning as a stylistic problem. If it is not a feasible bug right now, certainly it could be in future versions of the code. The lock on the game state should be held throughout the cleanup.

**Arbaro**

This program tracks the progress of an operation using a shared progress object. When it updates the progress, it checks to see if more than 1 percent progress has been made since the last time it was updated. If so, it sets the new progress. This is clearly intended to be atomic, since the operation of setting the progress depends on the result of the progress check. However, there is an atomicity bug. If another thread modifies the progress object between the check and the operation, the progress object enters an inconsistent state.

**TVSchedulerPro**

Our tool identified two atomicity bugs in TVSchedulerPro.

TVSchedulerPro maintains a list of devices capable of capturing video. Sometimes it needs to count the number of active devices in the system. However, the method that counts the devices has an atomicity bug. It acquires a lock on the list of capture devices to read the size of the list, but it releases the lock before it traverses the list. The size of the list could become stale if (for instance) a device is removed from the list. This leads to an access outside the bounds of the list, which will halt the program with an exception.

The second atomicity bug is in the system for setting server properties dynamically. When a property is needed, it first checks to see if the property has been defined. If it has not been defined yet, the property is set to a default value. However, an atomicity bug makes it possible for the property to be concurrently set after it is checked, but before the default value is assigned. Thus, the new value could be incorrectly overwritten by the default value.

**TuxGuitar**

The TuxGuitar application uses an object to represent a music sequencer. This sequencer contains a method that updates the sequencer forward one time step by managing a TickController object that stores the current time as well as information such as the tempo. This method should be atomic, so that it uses a consistent state of the TickController throughout the processing. However, there is an atomicity bug that allows the TickController to be modified by another thread, which results in unexpected behavior.

There are two instances of the pattern not representing an actual problem. In one case, the program uses a custom locking scheme, as described above. In the other case, a non-atomic method is unnecessarily declared as synchronized. This could be seen as a stylistic problem, but we categorize it as a benign pattern because it is completely harmless, and has little chance of causing problems in the future.

Our tool generates three warnings that do not correspond to an execution that exposes the pattern. An interprocedural path-sensitive analysis is required to detect that the pattern is not present.

**Jose**

This program contains a stylistic problem that involves an object that writes output data. This object is synchronized each time it writes, although some sequences of writes should be atomic. It is uncertain whether this atomicity is actually violated, but the stylistic issue is clear. In addition, there are three instances of encapsulated buffers whose synchronization could violate atomicity if the objects were shared.

There is a single warning that does not correspond to an instance of the pattern. Our call graph is too conservative to determine that the pattern is not present.

### 6. Related Work

This section describes work related to detecting and preventing atomicity violations. This can be divided into static and dynamic techniques.

Among the static techniques, Flanagan and Qadeer developed a type system for establishing atomicity in blocks specified by the programmer [5, 6]. Wang and Stoller use a static intraprocedural analysis to infer atomicity in the presence of non-blocking synchronization [23]. The above techniques require significant manual effort.

Von Praun and Gross use a fully automated technique to statically detect atomicity violations [20]. Their system in effect attempts to detect a different pattern than ours in Java code that is indicative of atomicity violations, so their approach is unlike and complementary to ours. In our experiments using the same benchmarks, our
future versions of the code. whose atomicity depends on assumptions that might not hold in unknown atomicity errors. Our tool also detects badly written code atomicity errors in those programs as well as several previously indeed correlate highly with atomicity violations. We checked over indicates that the pattern can indeed be detected efficiently and it does Our experience with several popular open source Java programs in-
for detecting occurrences of this pattern.
Our system infers which blocks of code must be atomic and detects violations in Java programs without requiring any specifications. This paper presents a system for automatically detecting atomicity violations (e.g., in the sor benchmark) that their tool could not. Also, it seems to us that our analysis is more efficient.
Among the dynamic techniques, there is work on systems that monitor the execution of Java programs and detect atomicity violations at runtime [4, 22]. These systems require programmer specifications of atomic blocks. Other techniques infer atomicity while running programs, either automatically [12, 13, 24] or using some basic annotations [14]. Dynamic techniques such as these rely on the paths exercised at runtime to expose a bug. In contrast, our static technique enables programmers to quickly run our system and immediately generate atomicity warnings.
Finally, there is work on implementing atomicity requirements using transactions [8, 9, 16]. These techniques approach the problem of atomicity from a different angle. Instead of checking that the programmer correctly used synchronization primitives, these systems provide language support for dynamically enforcing atomicity. Thus the programmer simply denotes which blocks are atomic and the language runtime enforces that requirement.

7. Conclusions

This paper presents a system for automatically detecting atomicity violations in Java programs without requiring any specifications. Our system infers which blocks of code must be atomic and detects violations of atomicity of those blocks. The key to our approach is the identification of a synchronization pattern that is highly likely to indicate a violation of atomicity and that can be detected efficiently using static analysis. The paper presents a static analysis for detecting occurrences of this pattern.

Our experience with several popular open source Java programs indicates that the pattern can indeed be detected efficiently and it does indeed correlate highly with atomicity violations. We checked over half a million lines of programs with our tool in a little over three minutes. Our tool successfully detects all the previously known atomicity errors in those programs as well as several previously unknown atomicity errors. Our tool also detects badly written code whose atomicity depends on assumptions that might not hold in future versions of the code.

References