A Static Analysis for Automatic Detection of Atomicity Violations in Java Programs

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 system also detects badly written code whose atomicity depends on assumptions that might not hold in future versions of the code.

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.

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 to prevent synchronization 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 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.

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.

Like other bug finding systems, our tool is neither sound nor complete. That is, our tool does not prove or refute atomicity. This is because the presence of the abovementioned pattern does not imply the presence of an atomicity violation, nor does the absence of the pattern imply the absence of atomicity violations. Moreover, because of the approximate nature of static analysis, our tool may falsely identify occurrences of the pattern as well as fail to identify occurrences of the pattern.

However, our experience with over half a million lines of real Java programs indicates that our tool is effective at identifying atomicity problems. It successfully detects all the previously known atomicity errors in those programs. This includes all the atomicity errors that were reported by other tools, including a *sound* type system [5, 6]

```
1  class Line {
2      Location start, end;
3      synchronized boolean contains(Location point) {
4          double r1=point.distanceTo(start);
5          double r2=point.distanceTo(end);
6          double c=start.distanceTo(end);
7          return ((r1+r2)<=1.001*c);
8      }
9  }
10
11 class Location {
12     double x, y;
13     synchronized double distanceTo(Location p) {
14         return Math.sqrt(
15                    Math.pow(p.x-x,2)
16                   +Math.pow(p.y-y,2)
17                );
18     }
19 }
```

Figure 1. An example code fragment illustrating a locking pattern that violates atomicity. While holding the lock on `this` in Lines 3-8, a lock on `point` is acquired and released twice.

and a static analysis tool [20]. In addition, our tool detects several previously unknown atomicity errors as well. Our tool also detects badly written code whose atomicity depends on assumptions that might not hold in future versions of the code.

Our tool is efficient, processing over half a million lines of code in a little over three minutes. Moreover, our experiments indicate that there are few false positive and they are easily detected. It took us less than an hour to manually analyze the warnings generated from about half a million lines of code to identify which ones are false positives and which ones are not. One reason we could identify the false positives quickly is that false positives often appear in groups. Once a false positive is found, it becomes clear that other warnings are false positives as well. For example, once we identify a set of lock objects as using a nonstandard locking idiom, any warning based on those objects is likely to be a false positive.

This paper makes the following contributions:

1. This paper identifies a synchronization pattern in Java code that is highly correlated with atomicity violations and that can be detected efficiently using a static analysis.
2. The paper presents a static analysis that efficiently detects the abovementioned pattern.
3. The paper presents a tool that implements the above static analysis. Our tool is efficient and analyzes over half a million lines of code in just over three minutes.
4. The paper describes our experience in using the above tool on over half a million lines of Java code from popular open source projects. Our tool uncovers a variety of problems including all the previously known and some previously unknown atomicity violations, and bad coding practices that could lead to atomicity errors in future versions of the code. The false positives are few and easily identified. It took us less than an hour to manually analyze the warnings generated from about half a million lines of code to identify the false positives.

The rest of this paper is organized as follows. Section 2 illustrates our approach with an example. Section 3 describes our approach for detecting atomicity violations. Section 4 presents a formal description of our efficient static analysis. Section 5 presents our experience using our tool to find atomicity violations in various Java programs. Section 6 discusses related work and Section 7 concludes.

Thread 1	Thread 2
compute `r1` (Line 4)	
	modify `point` from Loc1 to Loc2
compute `r2` (Line 5)	

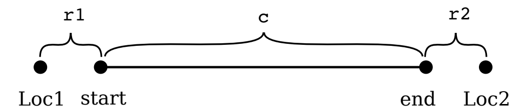

Figure 2. An example unserializable interleaving that demonstrates an atomicity violation of the `contains` method in Figure 1. The value `r1` is computed when `point` is Loc1, but the value `r2` is computed when `point` is Loc2. As a result, `r1+r2` is less than `c`, so `contains` returns `true` even though neither Loc1 nor Loc2 is contained in the `Line`. This does not correspond to any serial execution of `contains`.

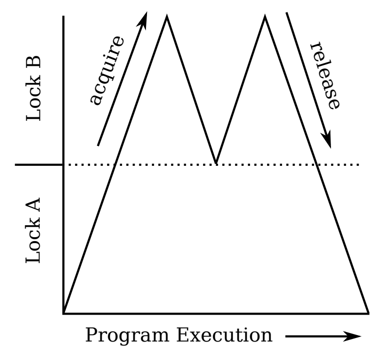

Figure 3. A graphical depiction of the locking pattern that causes atomicity violations. While the *context* lock A is held, the *witness* lock B is acquired and released twice.

2. Example

This section presents a motivating example.

2.1 An Atomicity Violation in Java Code

Consider the Java code in Figure 1, which is a simplified version of code from the ObjectDraw framework [2]. There are two classes, `Line` and `Location`. The `contains` method tests to see if the parameter `point` is contained within a `Line`. It computes `r1` and `r2`, the distances from `point` to each endpoint of the `Line`, then tests to see if the sum is within a certain tolerance of the length of the `Line`. Both classes are synchronized and intended to be used in the presence of concurrency. The `contains` method is intended to be atomic, so it acquires the lock on `this` before accessing the similarly synchronized `distanceTo` method twice on Lines 4-5. However, the `contains` method is not atomic. For example, the interleaving in Figure 2 is not serializable. In this interleaving, another thread modifies `point` (via one of `Location`'s methods) after `contains` computes `r1` but before it computes `r2`. If `point` changes from Loc1 to Loc2 then `contains` will return true although neither Loc1 nor Loc2 are in the `Line`. Hence there is no atomic serialization with the same result.

The method `contains` of Figure 1 has an atomicity violation. It first acquires the lock on `this`, then it calls `point.distanceTo` twice. Each call acquires and releases the lock on `point`. This is a pattern of synchronization like the one depicted in Figure 3, where `this` is lock A and `point` is lock B. To fix the atomicity violation, `contains` should explicitly acquire the lock on `point` for the duration of both calls to `distanceTo`:

```
synchronized(point) {
    r1=point.distanceTo(start);
    r2=point.distanceTo(end);
}
```

Locks in Java are reentrant, so the additional synchronization on the `point` object during the call to `distanceTo` is ignored.

2.2 Description of the Locking Pattern

Figure 3 depicts the synchronization pattern that resulted in an atomicity violation in the code of Figure 1. The pattern occurs when a lock B (the witness) is acquired and released twice while a *different* lock A (the context) is held.

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.

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 `contains` 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

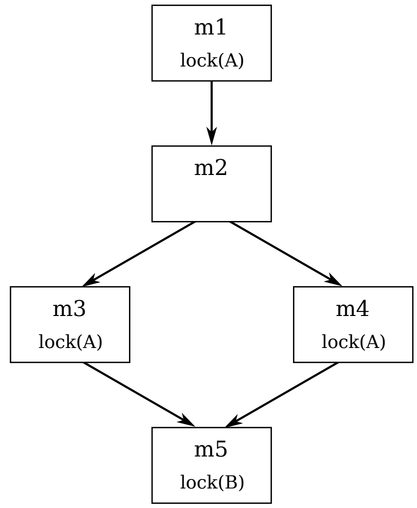

Figure 4. Example call graph with five methods. Method m1 calls method m2 while holding the lock on A. Our analysis shows that the lock on B is acquired and released twice while method m1 holds the lock on A.

Method	Locks Acquired	After Propagation
m1	A	A, B
m2	-	A, B
m3	A	A, B
m4	A	A, B
m5	B	B

(a)

Method	Locks Acquired Twice	After Propagation
m1	-	B
m2	A, B	A, B
m3	-	-
m4	-	-
m5	-	-

(b)

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.

`point`. For example, the following code clearly does not acquire and release the lock twice.

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

Our dataflow analysis tracks information about which locks have been acquired and released so far in the code. When we complete a method call to a synchronized method or exit a synchronization block, we mark the corresponding lock as having been acquired and released. If it is already marked, we mark it as having been acquired and released twice. For example, in Figure 1 the dataflow analysis has `point` marked as having been acquired and released zero times before Line 4, once after Line 4 and before Line 5, and twice after Line 5.

Note that the two acquisitions and releases of the lock on `point` need not be at distinct locations in the code. For example, if `start` and `end` are stored in an array called `endpoints` then the following code might be used.

```
for (int i = 0; i < 2; i++) {
    r[i]=point.distanceTo(endpoints[i]);
```

In this case, `point` is acquired and released twice, although each call to `distanceTo` takes place at the same code point. Our dataflow analysis traverses the loop twice before it stabilizes with `point` marked twice. However, we must be careful to ensure that the same lock is acquired and released both times. For example, consider the following code variation where two different points `point[0]` and `point[1]` are acquired and released in succession.

```
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[i]` 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 keyword `synchronized` 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

$$LE \quad ::= \quad v \mid this \mid LE.f \mid LE[c]$$

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.

In our analysis, locks are considered equal when they have the same syntax. Thus we use locking expressions as defined above to represent the locks themselves. In general, we consider two locks equal when their corresponding locking expressions are the same. Our experience indicates that this syntax-based alias detection works well, with no false positives as a result of imprecise alias detection discovered in our experiments. Therefore we choose to avoid the extra costs of a more complex alias analysis.

We model synchronization as a statement of the form `SYNC e sb` where `e` is a locking expression and `sb` is a synchronization block. The statement `SYNC * sb` refers to a synchronization block `sb` 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 $\bullet\mathtt{st}$ and $\mathtt{st}\bullet$. The set of all predecessors of a statement `st` is denoted $\mathit{pred}(\mathtt{st})$.

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 $A(m)$ denote the dataflow facts at the exit of method m, and let $A(\mathtt{sb})$ denote the dataflow facts at the exit of synchronization block `sb`. Let $\textsc{Callees}(\mathtt{st})$ denote the set of methods that may be called from the call site `st`. Let $\textsc{Calls}(m)$ denote the set of `CALL` statements in method m. Finally, let $\textsc{Methods}$ denote the set of all methods in the program.

4.2 Intraprocedural Dataflow Analysis

Our intraprocedural analysis detects the synchronization pattern described in Section 2 when it occurs within a method. We use a flow-sensitive forward dataflow analysis that tracks some facts about the synchronization history at each point in the program. In particular, our analysis maintains three sets:

- a set $R \subseteq$ *Locks* of previously *released* locks in the current static scope,
- a set $T \subseteq$ *Locks* of locks that were acquired and released *twice* in sequence (potential witnesses), and
- a set $P \subseteq$ *Locks* of locks that witness the full *pattern*.

Locks enter R 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

$$\langle R_1, T_1, P_1\rangle \sqsubseteq \langle R_2, T_2, P_2\rangle$$

if and only if

$$R_1 \subset R_2 \wedge T_1 \subset T_2 \wedge P_1 \subset P_2.$$

The join operator is defined such that

$$\langle R_1, T_1, P_1\rangle \sqcup \langle R_2, T_2, P_2\rangle = \langle R_1 \cup R_2, T_1 \cup T_2, P_1 \cup P_2\rangle.$$

The dataflow equations are presented below:

$$\begin{aligned} A(\bullet\mathtt{st}) &= \begin{cases} \langle\emptyset,\emptyset,\emptyset\rangle & \text{if } \mathtt{st} \in \mathit{entry}_m \\ \bigsqcup_{\mathtt{st}' \in \mathit{pred}(\mathtt{st})} A(\mathtt{st}'\bullet) & \text{otherwise} \end{cases} \\ A(\mathtt{st}\bullet) &= [\mathtt{st}](A(\bullet\mathtt{st})) \end{aligned}$$

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 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 `SYNC` statements look up the dataflow facts at the end of the synchronization block `sb`, denoted $\langle R_{sb}, T_{sb}, P_{sb}\rangle$. For a synchronization statement `SYNC e sb`, the set of previously released locks R is modified to include all locks released in `sb`, 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 `sb` as well as potential witnesses in `sb` 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 `SYNC * sb` 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.

Our analysis considers two locks equal when they have the same syntax, but that syntax might refer to different locks at different program points. The transfer function for an assignment `x := y` addresses this problem by assuming that the old value of `x` is lost and can not be equal to any further locks. More precisely, we remove from R all locks that depend on `x`. A lock e depends on `x` if it satisfies $\textsc{Depends}(e, x)$, which is defined at the the top of Figure 7. $\textsc{Depends}(e, x)$ 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. For example, incrementing a loop variable will remove from R all locks that index an array with that variable. This prevents our system from incorrectly assuming that distinct locks from across iterations are equal.

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 $\textsc{LocksHeld}(m)$, 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 `this` and formal parameters.

st	[st]($\langle R,T,P \rangle$)		
`SYNC e sb`	$\langle R \cup R_{sb} \cup \{e\},$	$T \cup (R \cap (R_{sb} \cup \{e\})),$	$P \cup ((P_{sb} \cup T_{sb}) \setminus \{e\})\rangle$
`SYNC * sb`	$\langle R \cup R_{sb},$	$T \cup (R \cap R_{sb}),$	$P \cup P_{sb} \cup T_{sb}\rangle$
`x := y`	$\langle R \setminus \{e \mid \textsc{Depends}(e,x) = true\},$	$T,$	$P\rangle$
all others	$\langle R,$	$T,$	$P\rangle$

for $A(\mathtt{sb}) = \langle R_{sb}, T_{sb}, P_{sb} \rangle$

(a)

st	[st]($\langle R,T,P \rangle$)		
`CALL` $e.m'(\bar{p})$	$\langle R \cup L,$	$T \cup (R \cap L),$	$P\rangle$

where $L = \bigcup_{m'' \in \textsc{Callees}(\mathtt{st})} \{\textsc{TranslateLock}_{\mathtt{st}}(x) \mid x \in \textsc{LocksHeld}(m'')\} \setminus \{*\}$

(b)

Figure 6. Transfer functions for (a) the intraprocedural analysis and (b) the first interprocedural extension

$$\textsc{Depends}(e,x) = \begin{cases} true & \text{for } e = x \\ \textsc{Depends}(y,x) & \text{for } e \neq x \text{ and } e = y.f \\ \textsc{Depends}(y,x) \vee \textsc{Depends}(i,x) & \text{for } e \neq x \text{ and } e = y[i] \\ false & \text{otherwise} \end{cases}$$

$$\textsc{LocksHeld}(m) = \{x \mid \mathtt{SYNC}\ x\ \mathtt{sb} \in m\} \setminus \{*\} \cup \left(\bigcup_{\substack{\mathtt{st} \in \textsc{Calls}(m) \\ m' \in \textsc{Callees}(\mathtt{st})}} \{\textsc{TranslateLock}_{\mathtt{st}}(x) \mid x \in \textsc{LocksHeld}(m')\} \setminus \{*\} \right)$$

For a method m that contains a statement `st` = `CALL e`$.m'(\bar{p})$ and for a locking expression x in the context of m',

$$\textsc{TranslateLock}_{\mathtt{st}}(x) = \begin{cases} \textsc{TranslateLock}_{\mathtt{st}}(y).f & \text{for } x = y.f \text{ and result} \in \textit{Locks} \\ \textsc{TranslateLock}_{\mathtt{st}}(a)[\textsc{TranslateLock}_{\mathtt{st}}(i)] & \text{for } x = a[i] \text{ and result} \in \textit{Locks} \\ \mathtt{e} & \text{for } x = \mathtt{this} \text{ and } \mathtt{e} \in \textit{Locks} \\ p_i & \text{for } x \text{ the parameter } i \text{ in } m' \text{ and } p_i \in \textit{Locks} \\ x & \text{for } x \text{ an integral constant} \\ * & \text{otherwise} \end{cases}$$

Figure 7. Equation for computing LocksHeld and definitions of Depends and TranslateLock

$$\textsc{Final}_P(n) = A(n)_P \cup \left(\bigcup_{\substack{s = \mathtt{SYNC\ e\ sb} \\ s \in \textsc{Syncs}(n)}} (\textsc{Final}_T(s) \cup \textsc{Final}_P(s)) \setminus \{\mathtt{e}\} \right)$$

$$\textsc{Final}_T(n) = A(n)_T \cup \left(\bigcup_{\substack{\mathtt{st} \in \textsc{Calls}(n) \\ m \in \textsc{Callees}(\mathtt{st})}} \{\textsc{TranslateLock}_{\mathtt{st}}(x) \mid x \in \textsc{Final}_T(m)\} \setminus \{*\} \right)$$

Figure 8. Equations for computing final sets of witnesses T and P for each method or synchronization block n

Before using the value of LocksHeld from another method, each lock needs to be translated to the current context. For example, the method `distanceTo` in Figure 1 locks `this`. In the method `contains`, that lock should be translated to the target of the call, `point`. Figure 7 defines the function $\textsc{TranslateLock}_{\mathtt{st}}(x)$, which translates locking expression x for the call site `st`. TranslateLock translates field and array accesses recursively. It translates `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 LocksHeld. For each method m, $\textsc{LocksHeld}(m)$ contains all locking expressions acquired in `SYNC` statements in m, as well as the translations of all of its callees' locks. We define Locks-Held 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 $\langle\{x, y\}, \emptyset, \emptyset\rangle$ and the final state of m' as $\langle\{y\}, \{y\}, \emptyset\rangle$. 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 represents 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 $\textsc{Calls}(n)$ for the case where n is a synchronization block. Let $\textsc{Syncs}(n)$ 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, $\langle T, P\rangle$, where T and P are as usual the sets of potential and actual witnesses, respectively. For a synchronization node n, define $A(n)$ as $\langle\emptyset, \emptyset, \emptyset\rangle$.

Figure 8 presents the equations for computing $\textsc{Final}_P$ and $\textsc{Final}_T$ for each node of the expanded call graph. $\textsc{Final}_T$ 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. $\textsc{Final}_P$ 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 s 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 $\textsc{Final}_P$ 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 generates 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, $\textsc{Final}_P(m)$ is the set of pattern witnesses of each method m. The final set of witnesses is then

$$\bigcup_{m \in \textsc{Methods}} \textsc{Final}_P(m).$$

The witness may not have been locked in m, so the same witness could appear in $\textsc{Final}_P$ 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 $\langle a, b\rangle \in \mathit{Locks} \times \mathit{Locks}$, where a was locked and released before b was locked. Thus the sets T and P are subsets of $\mathit{Locks} \times \mathit{Locks}$. 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.

Atomicity Violation

If a warning refers to a block of code that could plausibly be interleaved with unexpected results, it is considered an atomicity 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

st	$[\text{st}](\langle R,T,P\rangle)$		
`SYNC e sb`	$\langle R \cup R_{sb} \cup \{e\},$	$T \cup (R \times (R_{sb} \cup \{e\})),$	$P \cup ((P_{sb} \cup T_{sb}) \setminus \textsc{Fill}(e))\rangle$
`SYNC * sb`	$\langle R \cup R_{sb},$	$T \cup (R \times R_{sb}),$	$P \cup P_{sb} \cup T_{sb}\rangle$
`CALL` $e.m'(\bar{p})$	$\langle R \cup L,$	$T \cup (R \times L),$	$P\rangle$
all others	$\langle R,$	$T,$	$P\rangle$

for $A(\mathtt{sb}) = \langle R_{sb}, T_{sb}, P_{sb}\rangle$
and $L = \bigcup_{m'' \in \textsc{Callees}(\mathtt{st})} \{\textsc{TranslateLock}_{\mathtt{st}}(x) \mid x \in \textsc{LocksHeld}(m'')\}$

Figure 9. Transfer functions for detecting the pattern variant

$$
\begin{aligned}
\textsc{VarFinal}_P(n) &= A(n)_P \cup \left(\bigcup_{\substack{s=\mathtt{SYNC\ e\ sb} \\ s \in \textsc{Syncs}(n)}} (\textsc{VarFinal}_T(s) \cup \textsc{VarFinal}_P(s)) \setminus \textsc{Fill}(\mathtt{e}) \right) \\
\textsc{VarFinal}_T(n) &= A(n)_T \cup \left(\bigcup_{\substack{\mathtt{st} \in \textsc{Calls}(n) \\ m \in \textsc{Callees}(\mathtt{st})}} \{\textsc{TranslateLock}_{\mathtt{st}}(\langle x,y\rangle) \mid \langle x,y\rangle \in \textsc{VarFinal}_T(m)\} \setminus \textsc{Fill}(*) \right) \\
\textsc{TranslateLock}_{\mathtt{st}}(\langle x,y\rangle) &= \langle \textsc{TranslateLock}_{\mathtt{st}}(x), \textsc{TranslateLock}_{\mathtt{st}}(y)\rangle \\
\textsc{Fill}(x) &= \mathit{Locks} \times \{x\} \cup \{x\} \times \mathit{Locks}
\end{aligned}
$$

Figure 10. Equations for computing final sets of witnesses for the pattern variant, along with the helper function FILL

Benchmark	Lines of Code	Total Warnings	Errors and Potential Errors: Atomicity Violations	Errors and Potential Errors: Stylistic Problems	False Positives: Benign Pattern	False Positives: Non-Pattern	Time (s)
elevator	523	0	0	0	0	0	1.3
tsp	706	0	0	0	0	0	1.6
sor	17690	9	3	0	6	0	5.6
ObjectDraw	5637	3	2	1	0	0	2.7
Risk	10735	1	0	1	0	0	5.9
Arbaro	13760	1	1	0	0	0	6.6
TVSchedulerPro	30857	2	2	0	0	0	16.2
FreeMind	65161	0	0	0	0	0	23.1
TuxGuitar	96035	6	1	0	2	3	40.3
Jose	145993	5	0	4	0	1	48.8
JFreeChart	217354	0	0	0	0	0	40.9
TOTAL	604451	27	9	6	8	4	193.0

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.

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 synchronization pattern is used, although it may not lead to a violation of atomicity. For example, an object might not need correct synchronization because it is protected by a lock on a different object. However, the presence of incorrect synchronization on the encapsulated 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. Furthermore, there are ways to reduce the size of this category. Programmers can use annotations to tell our system that some locks are benign. With knowledge of the benign patterns, we could use heuristics to filter out these warnings. For example, locks that are held for most of the program's execution are probably not intended to be atomic blocks. Implementing such filters is future work.

Non-Pattern

Sometimes our tool produces warnings that do not correspond to instances 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. Additionally, 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 variables 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 were methods that were intended to be atomic but do not ensure atomicity through synchronization. There were 6 warnings with questionable style, which could lead to problems in future versions of the code. False positives accounted for the other 12 warnings generated. As expected, the analysis is efficient, processing over half a million lines of code in a little over three minutes.

It took us less than an hour to manually analyze the 27 warnings generated from about half a million lines of code and identify which ones are errors or potential errors and which ones are false positives. One reason we could identify the false positives quickly is that false positives often appear in groups. Once a false positive is found, it becomes clear that other warnings are false positives as well. For example, once we identify a set of lock objects as using a nonstandard locking idiom, any warning based on those objects is likely to be a false positive.

We also built the analysis of the pattern variant as described in Section 4.6 into our tool and ran it on all benchmarks. The variant analysis 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 investigation 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 determined that three of these warnings represent actual atomicity violations. Two atomicity violations are detected in the `addAll` methods of a data structure. These methods take a `Collection` as a parameter but they do not synchronize the parameter. Thus, changes to the `Collection` during execution of the `addAll` methods violates 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. We were not able to determine for certain whether this atomicity could be violated during execution, 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

Benchmark	Additional Warnings
elevator	0
tsp	0
sor	4
ObjectDraw	5
Risk	1
Arbaro	0
TVSchedulerPro	1
FreeMind	0
TuxGuitar	1
Jose	7
JFreeChart	0
TOTAL	19

Figure 12. Number of additional warnings gathered by running the analysis to detect the pattern variant.

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 tool detected 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

[1] Arbaro. http://sourceforge.net/projects/arbaro/.

[2] K. Bruce, A. P. Danyluk, and T. P. Murtagh. *Java: An Eventful Approach.* Prentice Hall, 2005. http://eventfuljava.cs.williams.edu/.

[3] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP Symposium on Objects and Databases*, August 1995.

[4] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Science of Computer Programming 71*, 2008.

[5] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Programming Language Design and Implementation (PLDI)*, June 2003.

[6] C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for Java. *Transactions on Programming Languages and Systems (TOPLAS) 30(4)*, July 2008.

[7] FreeMind. http://sourceforge.net/projects/freemind/.

[8] T. Harris and K. Fraser. Language support for lightweight transactions. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2003.

[9] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. *Principles and Practice of Parallel Programming (PPoPP)*, June 2005.

[10] JFreeChart. http://sourceforge.net/projects/jfreechart/.

[11] Jose. http://sourceforge.net/projects/jose-chess/.

[12] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

[13] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-aid: Detecting and surviving atomicity violations. In *International Symposium on Computer Architecture (ISCA)*, June 2008.

[14] B. Lucia, L. Ceze, and K. Strauss. Colorsafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *International Symposium on Computer Architecture (ISCA)*, June 2010.

[15] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Compiler Construction (CC)*, April 2003.

[16] M. F. Ringenburg and D. Grossman. Atomcaml: First-class atomicity via rollback. *International Conference on Functional Programming (ICFP)*, September 2005.

[17] Risk. http://sourceforge.net/projects/risk/.

[18] TuxGuitar. http://sourceforge.net/projects/tuxguitar/.

[19] TVSchedulerPro. http://sourceforge.net/projects/tvschedulerpro/.

[20] C. von Praun and T. Gross. Static detection of atomicity violations in object-oriented programs. *Journal of Object Technology 3(6)*, June 2004.

[21] C. von Praun and T. Gross. Object race detection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[22] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering 32(2)*, February 2006.

[23] L. Wang and S. D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. *Principles and Practice of Parallel Programming (PPoPP)*, June 2005.

[24] M. Xu, R. Bodk, and M. D. Hill. A serializability violation detector for shared-memory server programs. *Programming Language Design and Implementation (PLDI)*, June 2005.