Static Detection of Atomicity Violations in Object-Oriented Programs

Christoph von Praun and Thomas R. Gross

Laboratory for Software Technology
ETH Zürich
8092 Zürich, Switzerland

Abstract. Violations of atomicity are possible sources of errors in parallel programs. A violation occurs if the effect of a method execution depends on the execution of concurrent threads that operate on the same or overlapping parts of a shared data structure. All accesses to shared data are assumed to be ordered through synchronization, hence common techniques for data race and deadlock detection are not able to find such errors.
We have developed a static analysis that infers atomicity constraints and identifies potential violations. The analysis is based on an abstract model of threads and data. A symbolic execution tracks object locking and access and provides information that is finally used to determine potential violations of atomicity. We provide a detailed evaluation of our algorithm for a number of Java applications. Although the algorithm does not guarantee to find all violations of atomicity, our experience shows that our method is efficient and effective in determining several known synchronization problems in a set of application programs and the Java library. The problem of overreporting that is commonly encountered due to conservatism in static analyses is moderate.

1 Introduction

The use of locks and access to shared data in parallel programs entail the risk of errors that are not known in sequential programming: A possible source of error are violations of atomicity. A violation occurs if the effect of a method depends on the execution order of concurrent threads that operate on the same or overlapping parts of a shared data structure. Such a scenario is possible even if the data structure is protected through synchronization and individual accesses do not constitute a data race.

Atomicity is commonly understood as a property that is associated with statements and methods. Hence the search for violations of atomicity typically investigates the structure of statements and interleavings of threads. Flanagan and Qadeer [5, 4], e.g., have developed a type system that verifies the atomicity of methods. The type checker associates atomicities at the level of statements and combines these atomicities based on Lipton's theory of left and right movers [6] to obtain atomicity information for statement groups and methods. This approach requires explicit information about the synchronization discipline and lock protection of shared variables that are typically provided by program annotations.

The goal of this work is to provide a fully automated whole program analysis that detects methods that violate atomicity and identifies the data structures and calling contexts that lead to the violation. We assume that programs are free from data races and that accesses to shared data are ordered through monitor-style synchronization.

Our approach is motivated by previous work of Artho et. al. [1] which analyzes the structure of locking in a program and infers consistency constraints (*view consistency*) from the program for sets of shared variables. Violations of this consistency model correspond to potential synchronization defects called *high-level data races*. The notion of high-level data races is similar to violations of atomicity although both concepts are incomparable and several important scenarios of atomicity violations are not covered by the definition of high-level data races [9].

A common observation in programs that exhibit high-level data races or atomicity violations is that shared data is accessed with incoherent synchronization: Consider the example of a bank account in Figure 1: method `update` is invoked by several concurrent `Update` threads. The shared variable `balance` is accessed under common lock protection and hence there is no data race. The structure of locking specifies that the lock associated with the `Account` instance protects *either* a read *or* a write of field `balance`. Method `update` applies this synchronization discipline, however it performs a read *and* a write and hence cannot be atomic.

```
class Account {
   int balance;
   synchronized int read() {
      return balance;
   }
   int update(int a) {
      int tmp = read();
      synchronized(this) {
         balance = tmp + a;
      }
   }
}
```

```
class Update extends Thread {
   void run() {
      Example.a.update(123);
   }
}

class Main {
   static Account a;
   static void main(String args[]) {
      a = new Account();
      new Update().start();
      new Update().start();
   }
}
```

Fig. 1. Example of an account class with non-atomic `update` method.

The example illustrates a common pattern of software defects where one thread first queries the state of some shared data structure and then relies on this information during the further execution while being oblivious to the fact that other concurrent threads could have changed the data structure in the meantime. More generally, such faults are subsumed under the term *atomicity violations*.

The detection of atomicity violations is a research issue and different type-based and dynamic methods have been proposed [5, 4, 1, 9]. We address this problem with a static whole program analysis that detects potential violations in Java programs.

The analysis is based on an abstract model of threads and data. A symbolic execution tracks object locking and access, and provides information about the synchronization discipline that is finally used to determine synchronization defects. This algorithm is neither sound, i.e., there can be underreporting, nor complete, i.e., there can be overreporting. However, the algorithm detects all cases where one thread reads a shared variable under lock protection that may consequently be modified by concurrent threads (hence the result of the read might become stale). Our experience shows that this property covers a large number of cases of atomicity violations that have been reported earlier [5, 4, 9] and that overreporting is moderate (Section 4).

The focus of this paper is on the *static* detection of atomicity violations; other parts of the static analysis (that are, e.g., also used for escape analysis and static race detection) are mentioned only as far as necessary and are described in more detail in [7] and [8].

2 Method consistency

Method consistency specifies an access discipline for shared variables. The access discipline is determined from the access behavior of methods and the usage of locks. Method consistency adopts concepts from *view consistency* [1] and extends it to accommodate the scope of methods as consistency criterion. A violation of method consistency indicates a potential violation of atomicity at the method level. In some cases, such violations of atomicity are undesirable and represent software faults.

The rationale of method consistency is to conjecture atomic treatment for a set of shared variables that are accessed in the dynamic scope of a method (*method view*). The execution of a method is atomic if there are no concurrent updates of variables in its method view.

The activities of threads are modeled by *lock views*. A *lock view* is a set of *variable/access* pairs that correspond to variable accesses of a thread t in the dynamic scope of a lock. The *access* component specifies if the variable is read (`r`), or updated (`w`), i.e., written or read and written. There is one entry per variable. The set of lock views of a thread t is specified as $L_t = \{l_1, ...l_n\}$. For the program in Figure 1, lock views correspond to the fields accessed inside the synchronized method `read` and the synchronized block in method `update`: $L_{\texttt{Update}} = \{\{\texttt{balance/r}\}, \{\texttt{balance/w}\}\}$.

The conjecture about sets of variables that should be treated atomically is given through method views m_i for each method i. The set of method views per thread t it $M_t = \{m_1, ..., m_n\}$. There are two entries per variable, namely a read and a write entry. The method views for the `Update` threads in the example are $M_{\texttt{Update}} = \{m_{\texttt{update}}\} = \{\{\texttt{balance/r}, \texttt{balance/w}\}\}$.

We need two concepts to define method consistency: (A) *view overlap* and (B) the *chain property*: (A) Two views v_i and v_j *overlap* if their intersection is not empty, i.e., $v_i \cap v_j \neq \emptyset$. (B) A set of views $\{v_1, ...v_n\}$ *forms a chain* with respect to a view v, if for all pairs of views v_i, v_j, of which at least one originates from a thread that is concurrent to the originating thread of v, holds $((v_i \cap v) \subseteq (v_j \cap v)) \vee ((v_j \cap v) \subseteq (v_i \cap v))$.

Method consistency exists, if, for all method views, the overlapping lock views form a chain. The concept of overlap serves to filter out irrelevant variables. The chain property detects lock usage scenarios that are susceptible to atomicity violations: e.g., a lock protects reads *or* updates of one variable or a lock protects different but overlapping sets of variables (see high-level data races [1]). If method consistency is violated, a potential violation of atomicity is detected.

In the example, there are multiple concurrent threads $t_{\texttt{Update}}$ with a method view $m_{\texttt{update}} = \{\{\texttt{balance/r}, \texttt{balance/w}\}\}$. All lock views in $L_{\texttt{Update}}$ overlap with $m_{\texttt{update}}$, but they do not form a chain, hence method consistency is violated for method `update`.

3 Static analysis

The static detection of atomicity violations is based on a whole program analysis and is done in three steps. First, an abstract model of threads and heap data is computed (Section 3.1). Then a symbolic execution of abstract threads infers information about locking and object accesses (Section 3.2). Finally, claims of atomicity are established and validated (or refuted) (Section 3.3).

3.1 Modeling threads and data

In multi-threaded Java programs threads correspond to the execution of the `main` method and the `run` methods of objects that implement the interface `Runnable`. A compiler can determine *abstract threads* based on the allocation sites of `Thread` objects; the call graph of such threads is rooted at the `run` method of the thread object or an associated `Runnable` object. In many cases, a compiler cannot determine the actual number of runtime instances that originate at a thread allocation site; in this case, conservative assumptions are made and multiple concurrent runtime threads are assumed.

Java employs a simple memory model: Objects are allocated on a global heap and object access is possible only through references issued at object creation time. This model facilitates the approximation of the runtime object structure in a heap shape graph (HSG) [7] at compile time. Nodes in the HSG correspond to *abstract objects* and represent individual runtime objects or sets of objects that are aliased. Edges represent points-to relations introduced through reference fields. The overall result of the shape analysis is a set of graphs rooted at class or thread nodes.

For the purpose of determining view consistency, the construction of the HSG determines if abstract objects are potentially accessed from multiple runtime threads. Accesses to fields of such *shared* abstract objects during the symbolic execution (Section 3.2) constitute entries in views.

3.2 Symbolic execution

This section gives a brief overview on the symbolic execution that computes approximations of method views and lock views (Section 2) at compile-time. Details and optimizations are discussed in [8].

The symbolic execution analyzes individual instructions along the control-flow and call structure of an abstract thread; for multi-threaded programs, the analysis treats each abstract thread in a sequence. Before the analysis branches into a method invocation, the object context that the called method operates in is determined: For each local reference variable of the callee, the abstract object to which it refers is determined. For objects that are accessible to multiple threads, the abstract objects correspond to nodes in the HSG. Hence methods are analyzed in object contexts, and object accesses during the symbolic execution can be attributed to the abstract objects they target. A single traversal of loops and recursion is sufficient to track accesses to abstract objects (each iteration would access the same set of abstract objects).

The structure of locking in Java allows the compiler to track abstract objects that are locked along the execution path in a stack. At object access sites, e.g., *getfield*, *putfield* bytecodes, information about the locked abstract objects, the accessed abstract object, and the accessed field are available. Hence views can be recorded similarly to the runtime procedure in [1]. In contrast to (runtime) views, the static analysis computes *abstract views* with respect to locked abstract objects; in particular, only accesses to objects that are potentially shared are recorded. In our current implementation, abstract views contain field symbols and we do not distinguish accesses to fields on different abstract objects.

3.3 Detecting atomicity violations

The procedure to detect violations of method consistency is based on abstract views and checks the criteria described in Section 2 (overlap, chain); accesses to final, volatile, and read-only fields are omitted from the views.

Method consistency is designed as an extension of view consistency. For languages like Java, where synchronization is often used at method boundaries, violations of view consistency [1] imply violations of method consistency. Wang and Stoller [9] note that view consistency and the absence of atomicity violations are incomparable. The same holds true for method consistency. Hence, according to the classification of *soundness* and *completeness* [3], our procedure to determine atomicity violations is *unsound* and *incomplete*.

An example for the unsoundness of our algorithm, i.e., an atomicity violation that is not detected, is given in Figure 2.

```
class Counter {
   int i;
   synchronized int inc(int a) {
      i += a;
      return i;
   }
}
```

```
class Main extends Thread {
   static Counter c;
   static void main(String args[]) {
      c = new Counter();
      new Main().start();
      new Main().start();
   }
   void run() {
      int i = c.inc(0);
      c.inc(i);
   }
}
```

Fig. 2. Example of underreporting: Although the lock views that overlap with the view of method `run` form a chain, the sequence of updates in the `run` method is not atomic and does not necessarily double the counter value.

Figure 3 illustrates an examples for the incompleteness of our algorithm, i.e., a report of a method inconsistency that does not correspond to a violation of atomicity.[1]

So far, the conceptual capabilities of method consistency have been discussed. Additional imprecision is added through the fact that static analysis relies in

[1] The initialization pattern used in the example may not work as expected under current Java memory semantics; the memory model is being revised [2].

```
class Map {
   Object[] keys, values
   Object[] values;
   boolean volatile init_done = false;
   void init() {
      if (!init_done)
      synchronized (this) {
         init_done = true;
         // update keys and values
      }
   }
   synchronized Object get(...) {
      // read keys and values
      return ...
   }
}
```

```
class MapClient extends Thread {
   static Map m;
   static void main(String args[]) {
      m = new Map();
      new MapClient().start();
      new MapClient().start();
   }
   void run() {
      // lazy initialization
      m.init();
       o = m.get(...);
       ...
   }
}
```

Fig. 3. Example of overreporting: The lock views of method `run` do not form a chain (lock protects reads *or* updates), hence method consistency is violated. However, the initialization of the map happens only once and the effect of method `run` is the same regardless of the thread interleaving.

many cases on a conservative approximation of the runtime situation. Abstract views might subsume a larger number of variables than actual runtime views due to infeasible control-flows or the inability of the static analysis to differentiate accesses to field variables in different object instances. The approximation of the static analysis to distinguish different object instances and to determine thread interference and data sharing can lead to reports that do not correspond to real violations of atomicity – hence a potential source of overreporting.

4 Experience

We have implemented the static analysis in a way-ahead Java compiler and use the GNU Java library version 2.96.

First, we verify if our analysis is capable do detect known violations of atomicity that correspond to synchronization defects. Our system determines atomicity violations that correspond to the scenarios of the `Account`, `StringBuffer`, `Vector` and `PrintWriter` classes in [4]. Moreover, non atomicity in the use of iterators for common collection classes like `Vector` and `Hashtable` that are discussed in [9] are detected. These classes provide explicit means to determine actual violations of atomicity at runtime (`ConcurrentModificationException`). We have also successfully checked several scenarios with high-level data races, e.g., the `Coordinate` example in [1].

Second, we look at several benchmark programs and determine potential violations of atomicity at the application scope: philo is a simple dining philosopher application, elevator a real-time discrete event simulation; mtrt is a multi-threaded raytracer, tsp a traveling salesman application, hedc a web meta-crawler, specjbb an e-commerce benchmark, and jigsaw a public domain web-server (version 1.0). All other benchmarks stem from the multi-threaded Java Grande Benchmark suite.

I/O facilities are often shared among threads and interaction sequences of individual threads are usually not atomic. We omit this common case and do not report violations of method consistency related to I/O library classes. Moreover, our current implementation does not account for accesses to arrays and hence atomicity violations that are due to thread interference on shared arrays may not be reported.

Our implementation partitions views according to the affiliation of fields with classes. This means that the overlap and chain properties are determined only among field variables that belong to the same class. This strategy is justified because OO design typically imposes consistency constraints on variables of the same class; moreover, violations of the chain property for unrelated variables are omitted. We have not encountered a real synchronization defect that is overlooked due to this partitioning of views.

Accesses that occur during the initialization of a shared object or data structure cannot participate in inter-thread interference that leads to violations of atomicity. Hence we chose that views should not account for accesses through `this` in the scope of a constructor and for accesses in the scope of initializer methods. This convention is practical to reduce the number of spurious reports but entails the potential of underreporting.

The first columns in Table 1 show the execution times of the symbolic execution (*symexe*) and method consistency checking (*cons*) and the *memory* requirements of the static analysis on a P4 (1.4GHz) PC. Overall, the analysis is practical for the reported programs. The duration of the symbolic execution depends on the precision of type and alias information to narrow polymorphism. Programs like jigsaw and specjbb use dynamic class loading and instancing, which is modeled conservatively in the compiler and hence leads to imprecision. hedc, where conservative assumptions must be made due to a large recursion in the callgraph, is also negatively affected by conservative assumptions.

program	*size*	*time[s]*	*time[s]*	*mem*	*reports*		*methods*	
	LOC	*symexe*	*cons*	*[MB]*	*app*	*lib*	*app*	*lib*
philo	81	0.3	0.1	1	0	0	0/4	0/26
elevator	528	0.4	0.2	3	0/0/1/0	0/1/0/0	4/16	2/43
mtrt	11298	1.5	1.1	5	0	0/3/0/0	0/43	3/227
tsp	706	0.4	0.2	1	0/1/0/0	0	1/14	0/39
hedc	27952	140.8	19.9	58	0/1/4/0	3/7/6/0	11/135	3/350
specjbb	31903	62.1	23.9	30	0/17/0/0	0/3/3/0	19/447	5/248
jigsaw	31596	357.0	18.4	34	0/19/2/2	2/4/1/0	17/471	3/276
mol	1402	0.4	0.3	2	0/1/0/0	0/2/0/0	6/28	0/24
ray	1972	0.5	0.3	3	0/1/0/0	0/2/0/0	3/45	0/26
monte	3674	0.6	0.3	3	0	0/1/0/0	2/71	0/68
crypt	1241	0.1	0.1	2	0	0	0/10	0/1
lufact	1627	0.1	0.1	2	0	0	0/15	0/1
series	967	0.1	0.1	2	0	0	0/10	0/1
sor	876	0.1	0.1	3	0	0	0/7	0/1
sparse	868	0.1	0.1	2	0	0	0/8	0/1

Table 1. Analysis characteristics and reports of atomicity violations.

Further columns in Table 1 characterize violations of method consistency that we found. Column *reports* specifies the number of method views that are found to be inconsistent with lock views. We report only the smallest method views that still exhibit violations; method views that are supersets of those reported would exhibit the same violations but would make it more difficult to identify the cause of the report. If interference is due to field variables that belong to the library classes, numbers are reported in category *lib*, otherwise in category *app*. A report contains information about (1) the methods that exhibit this view, (2) the class to which the fields in this view belong to, (3) the individual field variables in this view, (4) potentially interfering lock views (locked object or method and the set of variables that cause interference), and (5) the allocation sites of the object on which the interference occurs. For each entry in column *reports*, we partition the reports into *false/spurious/benign/harmful*:

False reports are due to the imprecision of the static analysis (e.g., if data is not shared but actually thread-local).

Spurious reports specify that violations of atomicity do not occur at runtime in the given usage context of a data structure due to higher level synchronization (e.g., through a protected encapsulating object or thread start/join; see also example in Figure 3).

Benign reports refer to situations where an atomicity violation at the method level is possible. Such situations are not uncommon and do not necessarily represent a synchronization fault. This is especially true for methods that are invoked at a high level in the caller hierarchy of a multi-threaded application with shared data. In many cases, violations of atomicity are a natural consequence of the use of shared variables for explicit inter-thread communication.

Harmful reports mean that a violation of atomicity may occur that may lead to unintended runtime behavior.

The individual assessment of reports can be difficult and requires precise information about the synchronization disciplines for the affected shared data structures, hence we use this classification schema as a guidance.

Column *methods* in Table 1 specifies the number of methods reported by the checker and the overall number of methods for which a view is registered. The reports contain only methods that (1) access variables in at least one subordinate lock view (view v_1 is subordinate to v_2 if v_1 occurs in the scope of v_2), and (2) that are at the lowest levels of the caller hierarchy. Aspect (1) suppresses reports of method that do not use synchronization during their execution but exhibit a method view that is conflicting with lock views. For those methods, we report their callers (one of those will make use of synchronization because we assume that there are no data races). Aspect (2) excludes the reporting of all callers of a method for which we determined a potential violation of atomicity (a method that calls a non-atomic method is not atomic either). If a method belongs to a library class, it is reported in category *lib*, else *app*.

Most of the smaller benchmarks share data in arrays, hence there are few or no classes that we consider for reporting. In elevator, there is one benign report for a shared data structure that represents the state of the simulated system and is repeatedly accessed by the top-level methods of the simulator threads. A spurious report concerns an instance of class `Vector` that is however used such that no concurrent modification can occur.

mtrt exhibits three spurious reports that concern `Vector` and `Hashtable` data structures used in the library; these data structures are initialized once and then read (the scenario is similar to Figure 3). tsp has one spurious report due to a lock scope that violates the chain property but actually executes without concurrency during the initialization of the program.

In hedc, four reports are false and correspond to execution scenarios that the compiler conservatively assumed due to imprecise type information. Similar to mtrt, several reports are spurious on shared collection classes where initialization and subsequent shared read are ordered. Some reports are benign, e.g., for variables that are used to communicate information between worker and controller thread; another benign report addresses methods that perform subsequent accesses to a shared thread pool.

In specjbb, 12 reports correspond to classes that represent database records, where fields are accessed independently and atomicity is only necessary at the level of individual fields, or explicitly ensured by the transaction logic that is implemented at the application level. Depending on the correctness criteria at the application level, these reports can be classified as spurious or benign. Three further reports are benign and concern shared data containers that hold database records.

We discuss two interesting reports for jigsaw. The first report addresses class `ClientState` that represents an element of a linked list of client connections. Its fields `prev` and `next` link the structure and are accessed independently from fields `idle` and `client` (lock views are disjoint). All fields are cleared when a connection is removed from the pool and hence the fields are combined to a method view, leading to a report that does not reflect a problem in the program. The second report concerns class `ResourceStoreManager` in Figure 4. Method `shutdown` intends to remove all entries from the store (map referenced through field `entries`) and prevent further insertions by setting the latch `closed`. Atomicity is violated for method `loadResourceStore` (the sequence `checkClosed` and `lookupEntry` is not atomic). An unfortunate schedule can lead to the situation that entries are added to a resource manager after method `shutdown` has executed.

```
class ResourceStoreManager {
   boolean closed = false;
   Map entries = new HashMap();
   synchronized void checkClosed() {
      if (closed)
         throw new RuntimeException();
   }
   ResourceStore loadResourceStore(..) {
     checkClosed();
     StoreEntry se = lookupEntry(...);
     return se.getStore();
   }
   ...
}
```

```
synchronized Entry lookupEntry(...) {
   Entry e = (Entry) entries.get(..);
   if (e == null) {
      e = new Entry();
      entries.put(..., e);
   }
   return e;
}
synchronized void shutdown() {
   while (...) {
      // remove all entries
   }
   closed = true;
}
```

Fig. 4. Violation of atomicity in class `ResourceStoreManager` of the jigsaw benchmark.

The benchmarks mol and ray share part of the code and both report violation for lock views with disjoint variable sets on class `JGFTimer`. There is indeed a notion of consistency among the variables that could be violated if method calls would be interleaved in a particular sequence. There is however an explicit runtime check that detects this situation and issues a warning.

So far, views are restricted to shared variables. We have experimented with a further restriction: reads are only entered into lock views if the value is exposed outside the lock scope or method (i.e, the value is returned from a `synchronized` method or assigned to a stack escaping object). This modification reduces the number of reports by around 30-50%, however some cases of high-level data races are not recognized any more.

5 Conclusions

Violations of atomicity at the level of methods are common in parallel programs; in some cases however, such violations are undesired and considered as synchronization faults. This work presents a practical automated approach to detect such violations through an efficient whole program analysis. The detection is based on a notion of consistency (method consistency) that is inferred from the usage of locks and the accesses to shared data. Violations of method consistency match common cases of critical atomicity violations. Hence, method consistency is useful to identify a certain class of synchronization defects.

The common use of concurrent programming languages makes static tools for the automated detection of synchronization defects increasingly important. The focus of our static analysis on objects that are potentially shared makes the analysis efficient. The narrowing of the consistency notion to variables of the same class and the compaction of method reports along the caller hierarchy result in a moderate number of reports for the programs we assessed. Most reports do not reflect actual program defects. However, some of those reports that do not reflect a fault in the context of the analyzed program shed light on a synchronization discipline that could be insufficient if the affected data structure is reused in a different context.

Acknowledgments

We thank Cyrille Artho for helpful discussions and Matteo Corti and Florian Schneider for their contributions to the compiler system.

References

1. C. Artho, K. Havelund, and A. Biere. High-level data races. In *Rec. Wkshp. Verificat. and Validat. of Enterprise Information Systems (VVEIS'03)*, Apr. 2003.
2. D. Bacon, J. Bloch, J. Bogda, C. Click, P. Haahr, D. Lea, T. May, J.-W. Maessen, J. Mitchell, K. Nielsen, and B. Pugh. The "Double-Checked Locking is Broken" Declaration. http://www.cs.umd.edu/˜pugh/java/memoryModel, 2000.
3. C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proc. Conf. (PLDI'02)*, June 2002.
4. C. Flanagan and S. Quadeer. A type and effect system for atomicity. In *Proc. Conf (PLDI'03)*, June 2003.
5. C. Flanagan and S. Quadeer. Types for atomicity. In *Intl. Workshop on Types in Language Design and Implementation (TLDI'03)*, Jan. 2003.
6. R. Lipton. A method of proving properties of parallel programs. *Comm. of the ACM*, 18(12):717–721, Dec. 1975.
7. E. Ruf. Effective synchronization removal for Java. In *Proc. Conf. (PLDI'00)*, pages 208–218, June 2000.
8. C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proc. Conf. (PLDI'03)*, June 2003.
9. L. Wang and S. Stoller. Run-time analysis for atomicity. In *Rec. Workshop on Runtime Verification (RV'03)*, July 2003.