JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 6
Special issue: ECOOP 2003 workshop on FTfJP

Static Detection of Atomicity Violations in Object-Oriented Programs

Christoph von Praun and **Thomas Gross**
Laboratory for Software Technology, ETH Zurich, Switzerland

Violations of atomicity are possible sources of errors in parallel programs. A violation occurs if the effect of a method depends on the execution of concurrent threads that operate on the same shared data. Such unwanted thread interference can occur even if access to shared data is ordered through synchronization, hence common techniques for data race 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 several Java programs. Although the algorithm does not guarantee to find all violations of atomicity, our experience shows that the method is efficient and effective in determining several known synchronization problems in a number of applications 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 and one possible source of such errors are violations of atomicity. A violation occurs if the effect of a method depends on the progress of concurrent threads that operate on the same shared data. Such a scenario is possible even if shared data are protected through synchronization and access is ordered (i.e., there is no data race).

Atomicity is commonly understood as a property of statements and methods. Hence the search for violations of atomicity typically investigates the structure of statements and the interleaving 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 with statements and combines these atomicities based on Lipton's theory of left and right movers [8] 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.

Cite this article as follows: Christoph von Praun, Thomas R. Gross: "Static Detection of Atomicity Violations in Object-Oriented Programs", in *Journal of Object Technology*, vol. 3, no. 6, June 2004, Special issue: ECOOP 2003 workshop on FTfJP, pp. 103–122, http://www.jot.fm/issues/issue_2004_06/article5

The goal of this work is to provide a fully automated whole program analysis that detects methods that may not execute atomically and identifies the data structures and calling contexts that lead to the violation. We assume that programs are free from data races and that access to shared data is ordered through monitor-style synchronization.

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 Program 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 is not atomic.

Program 1: Class `Account` with non-atomic `update` method.

```
class Account {
    int balance;

    synchronized int read() {
        return balance;
    }

    void update(int a) {
        int tmp = read();
        synchronized(this) {
            balance = tmp + a;
        }
    }
}

class Update extends Thread {
    static Account acc;

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

    void run() {
        acc.update(123);
    }
}
```

The example illustrates a common pattern of software defects where a 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. Such faults are subsumed under the term *atomicity violations*.

Our 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 scenario covers most cases of atomicity violations that have been reported earlier [5, 4, 14] and that overreporting is moderate (Section 4).

2 METHOD CONSISTENCY

Similar to view consistency [1], *method consistency* specifies an access discipline for shared variables. Method consistency accommodates the method scope as consistency criterion, i.e., a violation of method consistency indicates a violation of atomicity at the method level. In some cases, such violations 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.

Definition (lock view). A *lock view* is a set of $\langle$*variable, access*$\rangle$ pairs that model the accessed variables and the kind of access performed by a thread t in the dynamic scope of a lock. *access* specifies if the variable is read (r), or updated (u), 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 $L_t = \{l_0, ...l_n\}$.

Example (lock view). In Program 1, lock views correspond to the fields accessed inside the synchronized method `read` and the synchronized block in method `update`: $L_{\texttt{Update}} = \{l_{\texttt{read}}, l_{<\texttt{syncblock}>}\} = \{\{\langle \texttt{balance}, \text{r}\rangle\}, \{\langle \texttt{balance}, \text{u}\rangle\}\}$.

Definition (method view). A *method view* models the conjecture about sets of variables that should be treated atomically. There is one method view m_i for each method i. A method view contains two entries per accessed variable, namely a read and an update entry (there are always both entries, irrespective the kind of access performed by the method). The set of method views of a thread t is $M_t = \{m_0, ..., m_n\}$.

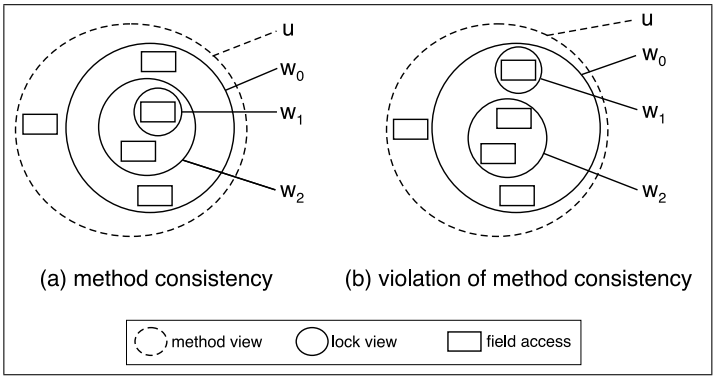

Figure 1: Illustration of method consistency.

Example (method view). In the example, the method views for the `Update` threads are[1] $M_{\texttt{Update}} = \{m_{\texttt{run}}, m_{\texttt{update}}\}$ where $m_{\texttt{run}} = \{\langle \texttt{balance}, \text{r}\rangle, \langle \texttt{balance}, \text{u}\rangle, \langle \texttt{acc}, \text{r}\rangle, \langle \texttt{acc}, \text{u}\rangle\}$ and $m_{\texttt{update}} = \{\langle \texttt{balance}, \text{r}\rangle, \langle \texttt{balance}, \text{u}\rangle\}$.

Definition (method consistency). We need two concepts to define method consistency:

- *View overlap.* Two views v_i and v_j *overlap* if their intersection is not empty, i.e., $v_i \cap v_j \neq \emptyset$.
- *Chain property.* A set of views $\{v_0, \ldots, v_n\}$ *forms a chain* with respect to a view v, if the set contains only a single element or for all pairs of non-empty views $w_i = v \cap v_i$, $w_j = v \cap v_j$, where at least one $v_{i,j}$ originates from a thread that is concurrent to the originating thread of v, holds $(w_i \subseteq w_j) \vee (w_j \subseteq w_i)$.

Method consistency is given 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.

Figure 1 illustrates the definition of method consistency using a method view v, lock views w_0, w_1, w_2, and a number of events that stand for runtime occurrences of read and write access to shared objects. In part (a) of the figure the lock views form a chain, i.e., they are nested. Part (b) shows a scenario where the lock views do not form a chain (w_1 and w_2 are not nested), hence method consistency is violated.

[1] We omit the method view for `read` because this method does not execute subordinate locking and does not call `java.lang.Object::wait`; provided that shared data access is synchronized through monitors, this method is not a candidate for an atomicity violation (details in Section 3.3).

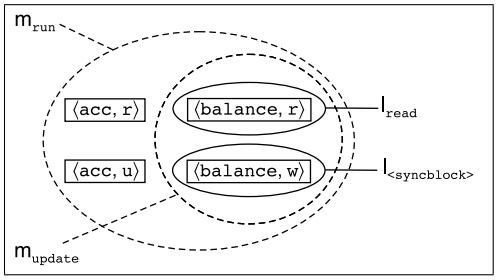

Figure 2: Method views and lock views for Program 1.

Example (method consistency). In Program 1, there are concurrent threads $t_{\texttt{Update}}$ with a method view $m_{\texttt{update}} = \{\langle \texttt{balance}, \mathrm{r}\rangle, \langle \texttt{balance}, \mathrm{u}\rangle\}$. All lock views in $L_{\texttt{Update}} = \{l_{\texttt{read}}, l_{\texttt{<syncblock>}}\}$ overlap with $m_{\texttt{run}}$ and $m_{\texttt{update}}$ but do not form a chain; hence method consistency is violated for methods `run` and `update`. Figure 2 illustrates this situation.

Program 2: Class `Account` with atomic `update` method.

```
class Account {
    int balance;

    synchronized int read() {
        return balance;
    }

    synchronized void update(int a) {
        int tmp = read();
        balance = tmp + a;
    }
}
```

Program 2 shows the corrected implementation of class `Account` where method `update` is atomic. If this implementation is used in the context of the `Update` threads in Program 1, method consistency exists: There is a single lock view in this program corresponding to method `update`, i.e., $L_{\texttt{Update}} = \{l_{\texttt{update}}\} = \{\{\langle \texttt{balance}, \mathrm{u}\rangle\}\}$; method `read` is only called in the scope of `update`, hence locking is reentrant in this invocation context of `read` and the analysis does not register a lock view for method `read`. The single lock view in $L_{\texttt{Update}}$ overlaps with the method view $m_{\texttt{update}}$ and, as there is only a single lock view, trivially forms a chain. Figure 3 illustrates this situation.

While method consistency is a property defined on a program execution (method and lock views contain field accesses), we describe in Section 3 how potential violations of method consistency can be determined by a static analysis.

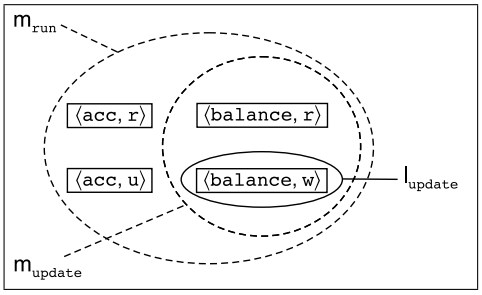

Figure 3: Method views and lock views for Program 1 with modified `Account` class (Program 2).

3 STATIC ANALYSIS

The static detection of atomicity violations is based on a whole program analysis that 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 access (Section 3.2). The two initial steps correspond to a more general analysis framework for multi-threaded Java programs that we describe in [13]; [13] reports how this general analysis framework is used for static data race detection. The target of the paper at hand is the detection of atomicity violations, which is done in the third step of the overall procedure: 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 `java.lang.Runnable`. A compiler can determine *abstract threads* based on the allocation sites of `java.lang.Thread` objects; the call graph of such threads is rooted at the `run` method of the thread object or an associated `java.lang.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) [9] at compile time. Nodes in the HSG denote *abstract objects* and represent a class, an individual runtime instance, or several instances that are aliased. Edges represent points-to relations introduced through reference fields. The result of the shape analysis is the HSG which is a set of graphs rooted at class or thread nodes. The HSG models all data that are accessible to different threads and hence are candidates for unwanted thread interference. The data model of the HSG is *flow-insensitive*, i.e., it approximates the runtime situation at any program point;

flow-insensitivity facilitates the analysis of multi-threaded programs.

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 [13].

The symbolic execution analyzes individual instructions along the call structure of an abstract thread; each abstract thread is treated separately. The analysis is *context-sensitive*, i.e., before the analysis branches into a method invocation, the object context in which the called method operates is determined: For each local reference variable in the callee, the abstract object to which it refers to is determined. Hence object access during the symbolic execution can be matched with its abstract target, i.e., a node in the HSG. A single traversal of loops and recursion is sufficient to track access to abstract objects because 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 on 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, it is sufficient to record only access to objects that are potentially shared. In our current implementation, The same field in different abstract objects is considered as the same variable (*field-based* analysis [15]). A more precise variable disambiguation would be possible, however there are several reasons that justify the simpler variant we present here:

- The abstraction errs on the conservative side because access events that *may* target the same instance at runtime target the same variable abstraction in the static analysis (one abstract object per class).
- In the programs at hand, most of the shared abstract objects are accessed by the same code in equivalent contexts, i.e., the lock and method views would be the same for different instances. Hence the overreporting introduced through the omission of the heap context information is minor.
- The implementation of the consistency checker and the reporting are simplified.

The analysis does not register lock views for lock operations that are found to be reentrant, i.e., an acquire operation on a lock that is already taken is ignored.

3.3 Detecting atomicity violations

Figure 4 shows the algorithm that determines potential violations of method consistency. The procedure is based on abstract views and checks the criteria described in Section 2 (overlap, chain). Input to the algorithm are sets of method views and lock views M and L that have been determined during the symbolic execution; access to final, volatile, and read-only fields is omitted from the views. The result of the algorithm is the set of violation reports R. The auxiliary functions *meth*, *class*, and *thread* take a view as argument and return the method (if view corresponds to a method), the class (views are partitioned according the affiliation of fields with classes, see below), and the thread that exhibits the view.

The algorithm operates along three phases:

1. Views are reduced and partitioned to limit the scope of search for subsequent phases: Fields that are *shared read-only* (heuristic used here: fields that are only assigned through `this` in the constructor) cannot bear interference in the form of atomicity violations. In addition, views of methods that do not execute synchronization actions in their dynamic scope are pruned from M because they are atomic, provided that the program is free from data races. Method *partition_views* 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 object-oriented design typically imposes consistency constraints on variables of the same class; moreover, spurious reports due to a 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.

2. Method consistency is assessed among the method and lock views according to the overlap and chain criteria. Method $overlap(u, L)$ returns a set of views form L that overlap with the view u. Method *check_chain* returns a set of potentially interfering lock views $ilv \subseteq L$ that violate the chain property with respect to a method view u.

3. The reporting is aggregated in the third phase of the algorithm. Assume some method m is found to violate method consistency; naturally, all callers of m will also violate method consistency. Hence, for a specific violation; only the lowermost method in the caller hierarchy is reported.

A report of a potential violation of method consistency specifies the following information:

- *meth*: the method that exhibits the critical method view u.
- *fields*: the field in the critical method view u.

$M = \langle$all method views$\rangle$;
$L = \langle$all lock views$\rangle$;
$R = \emptyset$;

method_consistency_analysis()

  /* phase 1: narrow views */
  readonly_fields $= \{f : \nexists u \in L : \langle f, \mathrm{u}\rangle \in u\}$;
  $\forall u \in L \cup M$:
    `if` $(\langle f, \_\rangle \in u \wedge f \in$ *readonly_fields*$)$
      $u = u - \{\langle f, \mathrm{u}\rangle, \langle f, \mathrm{r}\rangle\}$;
  $\forall u \in M$:
    `if` (*meth*(u) does not have subordinate locking)
      $M = M - \{u\}$;
  partition_views();

  /* phase 2: assess method consistency */
  $\forall u \in M$:
    $olv =$ *overlap*(u, L);
    $ilv =$ *check_chain*$(olv,$ *thread*$(u))$;
    `if` $(ilv \neq \emptyset)$
      $R = R \cup \{$`new` *Report*$($*meth*$(u),$ *class*$(u), u, ilv)\}$;

  /* phase 3: aggregate reporting */
  $\forall r \in R$:
    `if` $(\exists s \in R - \{r\} : s.$*fields* $= r.$*fields* $\wedge$
                $s.$*meth* is above $r.$*meth* in the caller hierarchy)
      $R = R - \{s\}$;

Figure 4: Algorithm for determining violations of method consistency.

.

- *class*: the class to which the fields in the method view u belong to.
- *olv*: set of lock views $\{l_0, ...l_n\}$ that overlap with the method view of m but do not form a chain.

Method consistency is designed as an extension of view consistency [1]. For languages like Java, where synchronization is used frequently at method boundaries, violations of view consistency imply violations of method consistency. Wang and Stoller [14] note that view consistency and the absence of atomicity violations are incomparable. The same holds true for method consistency: Our procedure to determine violations of atomicity at the method level is *unsound* and *incomplete* [3].

An example for the unsoundness of our algorithm, i.e., an atomicity violation that is not detected, is given in Program 3. Although there is only a single lock view $l_{\texttt{inc}}$ that overlaps with the method view $m_{\texttt{run}}$ (see Figure 5), and hence the chain property is trivially satisfied, the sequence of updates in the `run` method is not atomic and does not necessarily double the counter value.

Program 3: Example of underreporting.

```
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);
    }
}
```

Program 4 illustrates an example for the incompleteness of our algorithm, i.e., a report that does not correspond to a violation of atomicity: The lock views $l_{\texttt{init}}$ and $l_{\texttt{get}}$ do not form a chain (lock protects reads *or* updates), hence method consistency is violated (see Figure 6). However, the initialization of the map happens only once and the effect of method `run` is the same regardless of the thread interleaving. The general reason for this imprecision is that our analysis assumes that all control flow paths are feasible for all threads (which is does not hold for the example at hand).

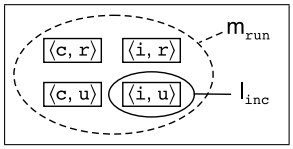

Figure 5: Method view and lock view for Program 3.

So far, the conceptual capabilities of method consistency have been discussed. Additional imprecision is added through the fact that static analysis relies in 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 access to field variables of different 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 [6] (version 2.96).

First, we verify if our analysis is able to detect known violations of atomicity that correspond to synchronization defects. Our system determines atomicity violations that correspond to the scenarios of the `Account`, `java.lang.StringBuffer`, and `java.io.PrintWriter` classes in [4]. Moreover, non atomicity in the use of iterators for common collection classes like `java.util.Vector` and `java.util.Hashtable` that are discussed in [14] are detected. These classes provide explicit means to determine actual violations of atomicity at runtime (`java.util.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 and application 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 [11], tsp a traveling salesman application, hedc a web metacrawler [12], specjbb [10] an e-commerce benchmark, and jigsaw an open source webserver (version 1.0alpha5) [16]. All other programs stem from the multi-threaded Java Grande Benchmark suite [7].

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 array access and hence atomicity

Program 4: Example of overreporting.

```
class Map {
    Object[] keys, values;
    boolean volatile init_done = false;

    void init() {
        if (!init_done)
            synchronized (this) {
                if (!init_done)  {
                    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() {
        m.init();  // lazy initialization
        o = m.get(...);
        ...
    }
}
```

violations that are due to thread interference on shared arrays are not 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 object-oriented 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 the partitioning of views.

Access that occurs during object initialization cannot participate in inter-thread interference that leads to violations of atomicity. Hence we chose that views should not account for access through `this` in the scope of a constructor and for access in the scope of initializer methods. This convention is practical to reduce the number of spurious reports but entails the potential of underreporting. A conservative and

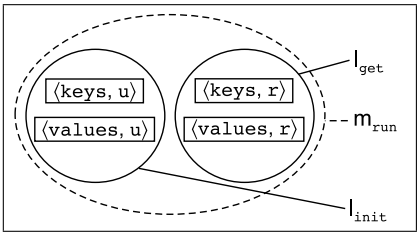

Figure 6: Method view and lock views for Program 4.

more precise algorithm to exclude access statements from critical interference is given in [13]; this analysis would remove the aforementioned source of underreporting.

The first columns in Table 1 show the size of the programs (lines of application code), the execution times of the symbolic execution (*symexe*) and method consistency checking (*cons*), and the *memory* requirements of the static analysis on a Pentium 4 (1.4 GHz) 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*	*symexe*	*cons*	*mem*	*reports*		*methods*	
	[LOC]	*[s]*	*[s]*	*[MB]*	*app*	*lib*	*app*	*lib*
philo	81	0.3	0.1	1	0	0	0/4	0/24
elevator	528	0.4	0.2	3	0–0–1–0	0–1–0–0	2/16	2/32
mtrt	11298	1.5	1.1	5	0	0–3–0–0	0/43	3/207
tsp	706	0.4	0.2	1	0–1–0–0	0	1/14	0/27
hedc	27952	140.8	19.9	58	0–2–3–0	3–5–2–0	10/141	3/366
specjbb	31903	62.1	23.9	30	0–17–3–0	1–3–0–0	19/472	4/317
jigsaw	31596	357.0	18.4	34	0–19–2–2	1–3–0–0	16/474	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	0/38
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: Program and analysis characteristics and reports of atomicity violations.

Column *reports* in Table 1 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*. 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 Program 4).

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. An exemplary situation where non-atomicity is desirable are methods that call `java.lang.Object::wait`: The execution of this method suspends the current thread, expecting that other threads change a shared (condition) variable and signal the state change such the current thread can continue execution. More general, any explicit inter-thread communication through shared variables will lead to a violation of atomicity.

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 discipline of the affected shared data structure, hence we use the 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_0 is subordinate to v_1 if v_0 occurs in the scope of v_1), 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 programs 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 `java.util.Vector` that is however used such that no concurrent modification can occur.

mtrt exhibits three spurious reports that concern `java.util.Vector` and `java.util.Hashtable` data structures used in the library; these data structures are initialized once and then read (the scenario is similar to Program 4). 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, three 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 access to a shared thread pool.

In specjbb, 10 reports correspond to instances 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 reports are benign and concern shared data containers that hold database records.

We discuss two interesting reports for jigsaw. The first report addresses class `w3c.jigsaw.http.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 `w3c.tools.store.ResourceStoreManager` in Program 5. 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.

The programs mol and ray share part of the code and both report violation for lock views with disjoint variable sets on class `jgfutil.JGFTimer`. There is indeed a notion of consistency among the variables that could be violated if methods 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

Program 5: Violation of atomicity in class `w3c.tools.store.ResourceStoreManager` of the jigsaw application.

```
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;
    }
}
```

number of reports by around 30-50%, however some cases of high-level data races are not recognized any more.

5 RELATED WORK

Method consistency is motivated by previous work of Artho et. al. [1], which analyzes the structure of locking in a program and infers consistency constraints for sets of shared variables (*view consistency*). Violations of their 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 [14].

Burrows and Leino [2] identify local variables that hold copies of shared data; a potential error occurs if during the program execution, the local copy becomes

inconsistent with the original shared variable but is still assumed to hold the up-to-date value. The kind of errors that are detected (stale-value errors) resemble violations of method consistency. Their analysis is complementary to our work, because it detects errors that are not found by our work (our procedure does not inspect uses of local variables; also the error in Program 3 would be found by the stale-value analysis), and vice versa (the error in Program 5 would not be found by the stale-value analysis).

Flanagan and Qadeer [5, 4] follow a different approach to detect potential violations of atomicity and focus on the structure of statements and the possible interleavings of statements through multi-threading. They have developed a type system that verifies atomicity. Unlike our technique that regards methods as the unit of atomic execution, Flanagan and Qadeer conjecture atomicity only for synchronized methods and blocks (the error in Program 5 would not be found). In their work, the type checker associates atomicities at the level of statements and combines these atomicities based on Lipton's theory of left and right movers [8] to obtain atomicity information for statement groups and methods. This approach is modular and requires explicit information about the synchronization discipline and lock protection of shared variables. This information is provided through annotations, hence Flanagan and Qadeer's technique is not fully automated.

Wang and Stoller [14] propose a dynamic technique to detect atomicity violations that is based on [5, 4] but improves on the precision. The key observation is that there can be groups of transactions that the reduction-based algorithm in [4] reports as non-atomic that still behave atomically in all schedules and hence should not be reported. Instead of individual transactions, the algorithm of Wang and Stoller groups transactions and searches for specific unserializable access patterns in groups of transactions.

6 FUTURE WORK

Our current implementation has some limitations that could be addressed in future work:

Currently, interference and hence violations of atomicity are only detected for methods that operate on shared objects, not arrays however. Array access could be registered in the method views and lock views, similarly to object access. While we have observed for object access that it is sufficient to register the mere fields without information about the accessed instance (Section 3.2), array access would need to distinguish different target arrays, and possibly access to disjoint parts of the same array; otherwise, spurious overlap between lock views and method views could lead to significant overreporting. The symbolic execution is a good platform to distinguish access to different arrays (and also objects) that are not aliased according to our heap shape model.

Our method conjectures atomicity at the level of methods and basically any method that performs a lock or wait operation is considered as a candidate for which an atomicity violation may be reported. This simple and very general rule is the major source of overreporting and hence if regions with mandatory atomic execution were specified explicitly in the code, the focus of checking could be narrowed and overreporting could be mostly avoided. Note that – even if atomicity constraints are specified – our method would not be sound due to the potential of underreporting (see Program 3).

The identification of shared data, which are the sources of interference that may lead to atomicity violations, requires the whole program. Future work could adapt our approach of method consistency to the modular checking of data types that pretend to have transaction semantics, i.e., guarantees for atomic execution, at their public interface. A challenging aspect for such a modular analysis is to distinguish between objects that are only accessed behind such an interface, i.e., that belong to the representation of the data type, and other objects that are not.

7 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. Unlike other research that determines atomicity from the equivalence of thread schedules at the level of individual statement interleavings [4, 14], our technique assesses atomicity through the observation of interference on shared data. Our procedures uses the notion of method consistency that is inferred from the usage of locks and access 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 and precise. 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 the anonymous reviewers for their insightful comments, Cyrille Artho for helpful discussions, and Matteo Corti and Florian Schneider for their contributions to the compiler system.

REFERENCES

[1] C. Artho, A. Biere, and K. Havelund. High-level data races. In *Proceedings of Workshop Verification and Validation of Enterprise Information Systems (VVEIS'03)*, Apr. 2003.

[2] M. Burrows and K. R. M. Leino. Finding stale value errors in concurrent programs. Research Report 2002-004, Compaq SRC, 2002.

[3] C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'02)*, pages 234–245, June 2002.

[4] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'03)*, pages 338–349, June 2003.

[5] C. Flanagan and S. Qadeer. Types for atomicity. In *Proceedings of the International Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 1–12, Jan. 2003.

[6] GCJ: The GNU Compiler for the Java Programming Language. http://gcc.gnu.org/java.

[7] Java Grande Forum multi-threaded benchmark suite. http://www.epcc.ed.ac.uk/javagrande/.

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

[9] E. Ruf. Effective synchronization removal for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'00)*, pages 208–218, June 2000.

[10] SPEC JBB2000 Java Business Benchmark. http://www.specbench.org/osg/jbb2000.

[11] SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98.

[12] E. Stolte, C. von Praun, G. Alonso, and T. Gross. Scientific data repositories – designing for a moving target. In *Proceedings on the International Conference on Management of Data and Symposium on Principles of Database Systems (SIGMOD/PODS'03)*, pages 349–360, June 2003.

[13] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'03)*, pages 115–129, June 2003.

[14] L. Wang and S. Stoller. Run-time analysis for atomicity. In *Workshop on Runtime Verification (RV'03)*, July 2003.

[15] J. Whaley and M. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the Static Analysis Symposium (SAS'02)*, Sept. 2002.

[16] World Wide Web Consortium. Jigsaw: Open Source web server. http://www.w3.org/Jigsaw.

ABOUT THE AUTHORS

Christoph von Praun is a PhD student and research assistant at the Laboratory for Software Technology at ETH Zurich, Switzerland.

Thomas R. Gross is a Professor at ETH Zurich and Head of the Laboratory for Software Technology, Switzerland.