Static Detection of Atomic-Set-Serializability Violations

Nicholas Kidd
Thomas Reps
Julian Dolby
Mandana Vaziri

Technical Report #1623

October 2007

Static Detection of Atomic-Set-Serializability Violations [⋆]

Nicholas Kidd[1], Thomas Reps[1,3], Julian Dolby[2], and Mandana Vaziri[2]

[1] University of Wisconsin {kidd,reps}@cs.wisc.edu
[2] IBM T.J. Watson Research Center {dolby,mvaziri}@us.ibm.com
[3] GrammaTech, Inc.; Ithaca, NY; USA.

Abstract. Vaziri et al. [1] propose a *data-centric* approach to synchronization. The key underlying concept of their work is the *atomic set*, which specifies the existence of an invariant that holds on a set of fields of an object type. In addition, they formalize a set of eleven *data-access scenarios* that completely specify the set of non-serializable interleaving patterns that can lead to an atomic-set serializability violation of the expressed invariant.

We present an algorithm that uses state-space exploration techniques to statically detect atomic-set serializability violations. The key idea is that the data-access scenarios can be used as a property specification for a software model checker. We tested our technique on programs with known serialiability violations from the concurrency-testing benchmark created by Eytani et al. Of the ten programs analyzed, our tool reported eight atomic-set serializability violations, with seven of them being true bugs.

1 Introduction

A common convention for writing concurrent software is to make use of multiple threads of execution that communicate through shared memory. This programming paradigm is present in many popular languages, such as C, C++, Java, and C#. Multithreaded applications that communicate through shared memory typically employ locks (or mutexes) to synchronize accesses on the shared state. Without any synchronization, multithreaded programs are prone to have data races. In the classic sense, a *data race* (or race condition) occurs when two threads of computation may access the same memory location without synchronization, and one of them is a write. There have been many research papers that address finding this type of data race, including [2–5].

Recently, Artho et al. showed that programs free from the classical kind of data race can still exhibit anomalous behavior [6]. Because of this, they extended the notion of data race to cover low-level and high-level data races. A low-level data race describes the classical notion of a data race (defined above). A *high-level* data race occurs when an invariant that exists between two or more fields of an object is violated.

[⋆] Supported by ONR under grant N00014-01-1-0796 and by NSF under grants CCF-0540955 and CCF-0524051. The views expressed herein are not necessarily those of the NSF or ONR.

Both low-level and high-level data races can be classified as atomic-set serializability violations. In [1], Vaziri et al. define the notion of an atomic set. An atomic set specifies the existence of an invariant that holds on a set of fields F of an object type T. It is a weak specification of a data-structure invariant [7]; it declares that the programmer intends that, for each T object, some data-structure invariant holds on the fields in F, although it does not specify what that invariant is—only that it exists. Associated with atomic sets are *units of work*. A unit of work is a sequence of program statements in which the invariant expressed by the atomic set is allowed to be temporarily violated. Upon completion of a unit of work, the invariant is presumed to be reestablished. Vaziri et al. also define eleven *data-access scenarios*, which specify interleaved data-access patterns that can lead to a violation of the invariant associated with an atomic set. A program execution is not atomic-set serializable if, during the execution of a unit of work by a thread, it exhibits one of the scenarios.[4]

We present an algorithm and an implementation, Empire, that performs static detection of atomic-set serializability violations. Empire is designed to detect violations[5] on an object that is allocated at a user-specified program point (i.e., a static-allocation site). To do so, Empire requires a "property specification" from the user. The property specification consists of (i) a collection of field names F that constitute an atomic set for some class T, (ii) a static-allocation site, α, and (iii) a scenario that is to be checked.

Our decision to focus on a specific object type T and static-allocation site α is based on the following: (1) [1] assumes complete control over all of a program's source code. Today's software-engineering practices advocate reusable components, which makes this assumption difficult in practice. Thus, we devised a technique that can perform violation detection, while continuing to allow the programmer the ability to use existing libraries and frameworks. (2) We desired a technique that was amenable to the analysis of legacy applications. That is, to reason about the atomic-set serializability of a legacy application with the techniques from [1], the atomic sets for *every* class would need to be determined. However, with our technique, a program analyst is only required to determine the atomic sets for *one* class. This allows an analyst to first focus on mission-critical components before attempting to reason about the classes of lesser importance (e.g., those whose instances are never shared between threads). As a side benefit, this design enabled us to create a more lightweight tool. By focusing selectively on one (or a few) shared object(s), Empire is able to abstract away much of the program when performing violation detection. Our experimental results show that more than 95% of the program can be abstracted away via our model-reduction techniques.

The key insight behind our approach is that an atomic-set specification coupled with a scenario can be used as an input specification to a model checker. Specifically, Empire translates a concurrent Java program into a *communicating pushdown system* (CPDS) [8, 9], and translates the atomic-set/scenario pair

[4] We will use the term "scenario" to refer to a data-access scenario.

[5] We will use the term "violation" to refer to an atomic-set serializability violation.

into a *monitor process*, which runs concurrently with the CPDS processes that model the program proper. Once the translation is performed, the generated CPDS is fed into a CPDS model checker [9]. By performing a translation from a concurrent Java program into a CPDS, Empire is able to leverage past research on software model checking, specifically CPDS model checking, for the task of violation detection.

Our work makes the following contributions:

- We present an algorithm that applies model-checking techniques to the problem of static violation detection. A benefit of the atomic-set serializability formalism, which is the most liberal of several related notions of serializability (see §7), is that it subsumes the notion of data races, both high and low-level.
- We present a technique that can analyze legacy applications without source-code modification. Empire only requires the analyst to provide a property specification, which itself is modularized to only describe a single Java class.
- We present model-reduction techniques, which our experiments show to be effective. In many cases they reduce the program to be analyzed by over 95%.
- We report on an initial implementation of our techniques. Our experiments show that it is effective in violation detection.

The rest of this paper is organized as follows: §2 discusses a program that contains a violation. §3 presents the details of atomic sets. §4 presents background material on CPDSs. §5 present the analysis steps that translate a concurrent Java program into a CPDS. §6 presents our initial experiments. §7 covers related work.

2 High-Level Data Race

Before explaining the details of Empire, we begin with a short discussion that illustrates a high-level data race. In Fig. 1, on line 17, the program assigns to the shared variable `v` a new instance of the class `java.util.Vector`. Next, on line 18, the program adds an instance of the class `java.lang.Integer` to `v`. At this point, `v` has one element in it. Finally, the program creates and starts two threads, t_1 at line 20 and t_2 at line 26. t_1 removes all elements from `v` via a call to `v.clear()` (line 22). t_2 creates a copy of `v` (line 29), ensures that the copy is not empty (line 30), and then proceeds to print the first object from the copy to standard output (line 32).

The program in Fig. 1 can encounter the well-known high-level data race in `java.util.Vector`, first reported by Wang and Stoller [10]. In this case, the program can crash by throwing a `NullPointerException`. The exception is triggered when the fields of the vector `w`, created at line 29 by thread t_2, become "out of sync". The problematic scheduling that provokes the exception is as follows:

1. Thread t_1 runs:
   (a) t_1 calls the `Vector` constructor passing `v` as an argument (line 29). This sets the `Collection c` parameter on line 5 to be `v`.

   (b) In the `Vector` constructor, t_1 sets `this.elementCount` (i.e., `w.elementCount`) to be the number of elements in the input `Collection c` parameter (i.e., `v`), which is equal to 1 (line 6).
   (c) t_1 allocates a new `Object` array and assigns it to `this.elementData` (i.e., `w.elementData`). All elements of this array are initialized to `null` (lines 7-9).
2. Thread t_2 runs:
   (a) t_2 removes all elements from the shared `Vector v` via the call to `v.clear()` (line 22). This sets `v.elementCount` to 0 and nulls out the array `v.elementData`.
3. Thread t_1 runs:
   (a) t_1 now calls `c.toArray(elementData)` to copy elements from `c` (i.e., `v`) to `this.elementData` (i.e., `w.elementData`) (line 12). However, `v`'s `elementCount` field is 0, so no elements are copied (in particular, `w.elementData[0]` *remains* `null`).

After the above sequence of operations, t_2 terminates and t_1 is at line 30, about to check whether `w` is empty. The conditional branch at line 30 is a safety precaution that the programmer inserted because he does not want to raise an exception. However, because of the high-level data race, `w.elementCount` and `w.elementData` no longer have the intended relationship; in particular, `w.elementCount` is equal to 1, while `w.elementData` is an array of `null` pointers. Thus, the conditional branch succeeds and the program follows the true path, which subsequently leads to the dereference of a `null` pointer (line 32) and a `java.lang.NullPointerException` is raised.

The underlying problem is that there is an implicit relationship between the fields `elementCount` and `elementData` of class `Vector`. Specifically, `elementCount` is meant to hold the current number of objects in the array `elementData`. Due to the high-level data race, this invariant can be broken and the program can crash. While the program is trivial, it makes clear the difficulty in reasoning about concurrent programs. In particular, relationships can exist among certain sets of fields of an object, and these relationships are not stated explicity in the program's code. The existence of these relationships, and the lack of a means for specifying them, led Vaziri et al. to formulate the notion of an *atomic set* [1].

3 Atomic Sets

Atomic sets indicate that an (unspecified) invariant is intended to hold on a set of fields of an object. In Fig. 1, we have annotated the fields `elementCount` and `elementData` with the comment `/*atomic(vec)*/`. This comment illustrates that the fields belong to the atomic set `vec`.

During the execution of a method of class `Vector`, the two fields may become inconsistent due to the granularity of individual program statements. For example, in the `Vector` constructor shown on lines 5–13 in Fig. 1, the field `elementCount` is first initialized to the size of the input parameter. Next, `elementData` is assigned a newly allocated array, which is then filled with the contents of the input parameter. During the execution of these statements,

```
1  public class Vector {
2    /*atomic(vec)*/Object[] elementData;
3    /*atomic(vec)*/int elementCount;
4
5    public Vector(/*unitfor*/Collection c) {
6      elementCount = c.size();
7      elementData = new Object[
8          (int)Math.min((elementCount*110L)
9                    /100,Integer.MAX_VALUE)];
10     // Potential high-level data race when
11     // c is changed by v.clear() (line 22)
12     c.toArray(elementData);
13   }
14 }
15 public class C {
16   void main(String[] args) {
17     final Vector v = new Vector();
18     v.add(new Integer(1));
19
20     new Thread(new Runnable() {
21       public void run() {
22         v.clear(); // v.elementCount = 0
23       }
24     }).start();
25
26     new Thread(new Runnable() {
27       public void run() {
28         // allocation site α
29         Vector w = new Vector(v);
30         if( !w.isEmpty() )
31           // NullPointerException possible here
32           print(((Integer)w.get(0)).intValue());
33       }
34     }).start();
35 }
```

Fig. 1. A high-level data race in the `java.util.Vector` constructor from Java SDK 1.5.

`elementCount` and `elementData` become inconsistent. Vaziri et al. refer to such statements as a unit of work [1]. In Fig. 1, the entire constructor method is a unit of work for the atomic set `vec`. In [1], the default policy is that each method of a class is a unit of work for the atomic sets of that class.

Vaziri et al. also defined a way to dynamically extend the members of an atomic set through a new language construct: *unitfor*. In essence, when a particular parameter p of method `T.m(...,p,...)` is labeled with `unitfor`, it asserts that the value of p will contribute to whatever actions `T.m` performs to reestablish the (unspecified) data-structure invariant on the fields in the atomic set. Therefore, `T.m` needs to see a consistent value for p. For example, in Fig. 1, we have annotated the input parameter `Collection c` on line 5 with `/*unitfor*/`. Because `c` is used by the constructor to establish the invariant on fields `elementCount` and `elementData`, the constructor needs to see a consistent view for `c`. Therefore, it is necessary to include the fields of `Collection c` in the atomic set `vec` during the execution of the constructor.

Atomic sets have the nice property that violations involving members of an atomic set are characterized by eleven forbidden scenarios.[6] Moreover, each scenario can be specified by a regular language. The program in Fig. 1 violates Scenario 9. In particular, it is possible for the following sequence of operations to occur: $R_{t1}(v.elementCount)$, $W_{t2}(v.elementCount)$, $W_{t2}(v.elementData)$, $R_{t1}(v.elementData)$, where $R_{tn}(f)$ ($W_{tn}(f)$) denotes a read (write) by thread

[6] See [1] for a complete listing of all eleven scenarios.

Rule	Control flow modeled
$\langle p, n_1\rangle \overset{a}{\hookrightarrow} \langle p, n_2\rangle$	Intraprocedural edge $n_1 \rightarrow n_2$
$\langle p, c\rangle \overset{a}{\hookrightarrow} \langle p, e_f\ r\rangle$	Call to f from c that returns to r
$\langle p, x_f\rangle \overset{a}{\hookrightarrow} \langle p, \varepsilon\rangle$	Return from f at exit node x_f

Fig. 2. The encoding of a call graph's and CFG's edges as PDS rules. The action a denotes the abstract behavior of executing that edge.

tn to field f. This sequence of operations is an atomic-set serializability violation (violation), and results in the program crash described in §2. Our work is based on the observation that the eleven forbidden scenarios can be checked by reachability analysis on a communicating pushdown system.

4 Communicating Pushdown Systems

This section presents background definitions for both pushdown systems and communicating pushdown systems.

Definition 1. *A pushdown system (PDS) is a four-tuple $\mathcal{P} = (Q, Act, \Gamma, \Delta)$, where Q is a finite set of states, Act is a finite set of* actions, *Γ is a finite stack alphabet, and Δ is a finite set of rules of the form $\langle q, \gamma\rangle \overset{a}{\hookrightarrow} \langle q', u\rangle$, where $q, q' \in Q$, $a \in Act$, $\gamma \in \Gamma$, and $u \in \Gamma^*$. A configuration of $\mathcal{P}$ is a pair $c = \langle q, u\rangle$, where $q \in P$ and $u \in \Gamma^*$ is the stack contents. A set of configurations C is regular if for each $q \in Q$ the language $\{u \in \Gamma^* \mid \langle q, u\rangle \in C\}$ is regular. The PDS rules Δ define a (potentially infinite) transition system on the configuration space of $\mathcal{P}$.*

Without loss of generality, we restrict the right-hand-side of all PDS rules to contain at most 2 stack symbols [11].

For all $u \in \Gamma^*$, a configuration $c = \langle q, \gamma u\rangle$ can make a transition to a configuration $c' = \langle q', u'\,u\rangle$ if there exists a rule $r \in \Delta$ of the form $\langle q, \gamma\rangle \overset{a}{\hookrightarrow} \langle q', u'\rangle$. We denote this transition by $\xrightarrow{a}$ and extend it to $\xrightarrow{a_1 \cdots a_n}$ in the obvious manner. For a set of configurations C, we define $post^*(C) = \{c' \mid \exists c \in C, a_1 \cdots a_n \in Act^*, c \xrightarrow{a_1 \cdots a_n} c'\}$.

Because PDSs maintain a stack, they naturally model the interprocedural control flow of a thread of execution. The translation from a call graph and set of control-flow graphs (CFGs) into a PDS is shown in Fig. 2.

Definition 2. *A communicating pushdown system (CPDS) is a tuple $CP = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ of PDSs. The action set of CP is equal to the union of the action sets of the $\mathcal{P}_i$, along with the special action τ: τ has the property that for all $a \in \bigcup_1^n Act_i$, $\tau a = a \tau = a$. A global configuration of a CPDS CP is a tuple $g = (c_1, \ldots, c_n)$ of configurations of $\mathcal{P}_1, \ldots, \mathcal{P}_n$.*

For CPDSs, the reachability relation $\xrightarrow{a}$ is extended to global configurations as follows:

- $(c_1, \ldots, c_n) \xrightarrow{\tau} (c_1', \ldots, c_n')$ if there exists an index $1 \leq i \leq n$ such that $c_i \xrightarrow{\tau} c_i'$ and, for every $j \neq i$, $c_j = c_j'$.

- $(c_1, \ldots, c_n) \xrightarrow{a} (c'_1, \ldots, c'_n)$ if for $1 \leq i \leq n$, $c_i \xrightarrow{a} c'_i$.

For a set of global configurations G, we define $post^*(G) = \{g' \mid \exists g \in G, a_1 \ldots a_n \in Act^*, g \xrightarrow{a_1 \ldots a_n} g'\}$.

A CPDS models a multi-process message-passing system. The reachability relation captures a *rendezvous-style* means of synchronization. At first glance, global rendezvous might seem like a weakness of the CPDS system; however, pairwise synchronization (i.e., lock-based synchronization) is easily modeled in the CPDS. To do this, each PDS $\mathcal{P}_i$ is augmented to allow for any action not in Act_i to occur at *any* time. This can be achieved by augmenting $\mathcal{P}_i$ with rules that non-deterministically "fire" actions not used by $\mathcal{P}_i$ for any configuration. Specifically, a rule of the form $\langle q, \gamma \rangle \overset{a}{\hookrightarrow} \langle q, \gamma \rangle$ is generated for each state $q \in Q$, stack symbol $\gamma \in \Gamma$, and action a not in $Aisct$.

The goal is to determine if there exists a common string of actions in the *language* of each process of the CPDS, where a process consists of a PDS $\mathcal{P}$, along with its initial and final sets of configurations. This problem can be reduced to a reachability query in a CPDS: the model-checking algorithm for answering the reachability problem takes as input a set of PDSs, a set of initial configurations G, and a set of final configurations G', and checks whether $G' \cap post^*(G) = \emptyset$ holds. Because the action languages can, in general, be context-free languages and the problem of checking their intersection for emptiness is known to be undecidable, the CPDS reachability algorithm is only a *semi-decision* procedure. The semi-decision procedure may not terminate, but is guaranteed to terminate if there exists a finite-length sequence of actions, $a_1 \ldots a_n$, such that the following holds: $g \in G,\ g' \in G',\ g' \in post^*(G)$, and $g \xrightarrow{a_1 \ldots a_n} g'$ [8]. Additionally, in some cases, the semi-decision procedure can determine that there *does not* exist any common sequence of actions, and thus that G' is not reachable from G.

5 Violation Detection

Violation detection is performed in two phases. First, a concurrent Java program is abstracted into a program in Empire's lower-level modeling language (ESL) via Empire's front-end `j2esl`. Second, Empire's back-end, `esl2cpds`, translates the ESL program into a CPDS, which is then fed to the CPDS model checker.

5.1 Model Reduction

The `j2esl` front-end uses the WALA program-analysis infrastructure from IBM [12]. Specifically, `j2esl` makes use of several components of WALA's intermediate representation of Java programs, including the call graph, the procedure control-flow graphs (CFGs) associated with call-graph nodes, a Java class hierarchy, and a database of pointer-analysis results.

Because Empire performs violation detection on one static-allocation site at a time (α), it is necessary to disambiguate method invocations whose receiver instance may be the object allocated at α. To do this, `j2esl` uses a points-to analysis based on Milanova et al.'s object-sensitive points-to analysis [13] during call-graph construction. We found during our experiments that building a fully object-sensitive call graph did not scale. Instead, `j2esl` only specializes the call graph for allocation sites whose object types are either of class T or the type of one of the fields in F (recall that T and F are provided by the user via the property specification). We have found that selective object sensitivity provides enough precision to track reads and writes to α while retaining scalability.

`J2esl` first performs a series of model-reduction steps before generating an ESL program. Namely, it takes advantage of the fact that violation detection is performed one allocation site at a time. This allows `j2esl` to prune away much of the original program when generating an ESL program. For example, analyzing a simple "hello world" program in WALA loads 15,437 Java classes. Many of these classes come from the large `java.io` library. For both efficiency and debugging purposes, it is beneficial to prune out code irrelevant to violation detection. An additional benefit of pruning is that many of the locks are eliminated. The pruning approach used is to selectively grow the set of call-graph nodes that are necessary to accurately model the program. We do this in three steps.

Step 1 `J2esl` determines the set of call-graph nodes, N_α, whose CFGs include program statements that may read or write to a field $f \in F$. To do this, `j2esl` traverses the CFG of each call-graph node, checking to see if any statement may (transitively) read or write to a field $f \in F$ of α. During this process, an instruction that acquires or releases the mutex associated with α is considered to be a write. We define R_α to be the set of call-graph nodes backwards reachable from any node $n \in N_\alpha$. R_α represents the minimal set of methods that must be modeled accurately by Empire.

Step 2 `J2esl` next determines the set of allocation sites that, during program execution, might be dynamically included in the atomic set. Remember that an atomic set can be dynamically extended because of the *unitfor* annotations. The user has the ability to manually specify which object parameters to a method of class T are unitfor parameters, or `j2esl` can be instructed to assume that any object parameter to a method invoked on α is annotated with `unitfor`. To determine the unitfor objects, `j2esl` visits each call-graph node $r \in R_\alpha$. If the receiver of r may be α, then for each object parameter that is designated with unitfor for the method, the points-to database is queried to determine the static-allocation sites that might be dynamically included in the unit of work for α. We define this set as U_α. Finally, similar to Step 1 above, each CFG of the program's call graph is traversed to locate the set of call-graph nodes, N_{unitfor}, for methods that may read or write to the fields of the objects in U_α. Let R_{unitfor} be the set of call-graph nodes backwards reachable from N_{unitfor} and let $S = R_\alpha \cup R_{\text{unitfor}}$.

Step 3 Once S has been identified, `j2esl` traverses the CFG for each node $n \in S$ to determine the set of locks that may be acquired in n. This set of locks consists of the static-allocation sites that can be the receiver for an invocation of a Java `synchronized` method, or that can be used as the mutex for guarding a Java `synchronized` block. In each case, the static-allocation sites are determined by querying the points-to database. Let this set of locks be L.

Theorem 1. *The Java program that is the result of `j2esl`'s model-reduction steps is a sound approximation of the original Java program with respect to* *violation detection on the object of type T, allocated at static-allocation site α, and for the atomic set consisting of the fields in F.*

Proof (sketch): Our sketch relies on the soundness of of the call graph, CFGs, and the points-to database generated by the WALA analysis infrastructure. The CFG node for any Java statement that potentially reads or writes to the atomic set denoted by F for the object allocated at α will be in S. Thus, we only need to reason that the program statements of a CFG node $n' \notin S$ cannot cause a violation. First, the method represented by n' (and those transitively reachable from n' in the call graph) might acquire and release a lock. However, because of the model-reduction steps just described, we know that n' and its successors in the call graph have no effect on α, the fields $f \in F$, or the objects in U_α. Additionally, because of the syntactic nature of Java `synchronized` methods and blocks, any lock acquired must be released upon return from n' to n. Therefore, it is safe to ignore the synchronization that might have resulted from invoking n' because the overall status of the locks that are held remains unchanged upon return from n'. Second, the invocation of n' might cause exceptional control flow paths to be exercised due to an exception being triggered within n' (or methods transitively called from n'). The ESL program accurately models these paths because WALA safely models (i.e., overapproximates) the interprocedural control flow of the Java program being analyzed, including control flow for exceptions.

5.2 ESL Generation

After completing the model-reduction steps, `j2esl` generates an ESL program. An ESL program consists of a finite number of abstract memory locations, locks, and processes. The abstract memory locations model the fields F of the atomic set for static allocation site α. Only reads and writes are allowed to be performed on the abstract memory locations.

A Java thread is modeled by an ESL process, which consists of a set of (potentially recursive) functions, where one of the functions is designated as the process's entry point. `J2esl` finds all static threads in the program by checking each instruction in each CFG to see if it invokes the method `Thread.start()`. The points-to database is then queried to determine the static-allocation sites that are potential receivers of this method invocation. We denote this set of objects as E because they represent the *entry* points for the threads. Additionally, E includes a "fake" object that represents the initial thread created when the Java program begins execution in the `main` method. For each entry point $e \in E$, `j2esl` walks the call graph in a depth-first manner, translating the CFG for each reachable node n in the call graph into an ESL function. For each instruction in n, `j2esl` emits an appropriate ESL statement (see Tab. 1). If a Java instruction invokes a method whose call-graph node $n' \notin S$, the method invocation is ignored and the emitted ESL statement is a `skip` statement.

An ESL process's functions are abstractions of the methods that a Java thread might invoke. The ESL statements that make up a function model the (intraprocedural) control flow of a Java method. Each statement is either a `goto`, `choice`, `skip`, `call`, `return`, `read x`, `write x`, `lock l`, `unlock l`, `unit-begin`, `unit-end`, or `start p`. Tab. 1 presents some example Java instructions and their corresponding ESL statements. The ESL statement `start p` starts the ESL process denoted by `p`. This is used to model the fact that when a Java program begins, only one thread is executing the `main` method. Child threads must be started by an already running thread. An ESL process that models a child thread cannot begin its (abstract) execution until it is started by an already running process.

`x = o.f`	`read` f	$\alpha \in \text{Pts}(o)$
`o.f = x`	`write` f	$\alpha \in \text{Pts}(o)$
`synchronized(o){...}`	`lock` o;. . . ;`unlock` o	
`o.start()`	`start` o	`Thread.start()` is invoked
`o.u()`	`unit-begin;call` u; `unit-end`	$\alpha \in \text{Pts}(o)$, `u()` is a unit of work
`o.m()`	`call m`	

Table 1. Example Java instructions, their corresponding ESL statements, and the condition necessary to generate the ESL statement.

ESL locks model the Java mutexes used for synchronization. They are reentrant locks because Java mutexes are reentrant. An ESL program is restricted to a finite number of locks. To meet the finiteness requirement, `j2esl` creates an ESL lock for each static-allocation site that is in L. Thm. 1 ensures that if the original program can have a violation, then the reduced program is also vulnerable to the same violation. However, the translation from the reduced program to ESL is unsound when one ESL lock models many concrete Java mutexes. In such a situation, a state change of an ESL lock is, in effect, performing a strong update of many concrete Java objects simultaneously. This can cause the ESL program to underapproximate the Java program's actual set of behaviors. However, our experiments show that Empire is an effective violation detector.

5.3 CPDS Generation

Empire's back-end, `esl2cpds`, translates the generated ESL program into a CPDS. This consists of translating the ESL program into a message-passing system, where each message conveys the setting or querying of some global property of the CPDS. In terms of violation detection, the global state that must be kept track of includes: (i) the status of every shared lock, (ii) whether an ESL process is in a unit of work, (iii) all reads and writes to members of the atomic set, and (iv) the *monitor process* that models the scenario. We next describe how `esl2cpds` generates PDSs to model each of these components.

Lock process Empire tracks the state of each shared lock by creating a separate *lock process*. The PDS that models a mutex m changes its state from open to locked and vice versa when it receives a lock and unlock message from another ESL process. The PDS for m uses the stack to maintain a count of the number of times the lock is acquired by an ESL process, as well as the name of the locking process. This is necessary because a Java thread can acquire a lock that it currently holds multiple times (and must subsequently release that lock an equal number of times). The following PDS rules define a template that can be instantiated with the name of an ESL process:

$$\langle q, \mathit{open}\rangle \xhookrightarrow{\mathit{lock}_p} \langle q, \mathit{lock}_p\ \mathit{open}\rangle \qquad \langle q, \mathit{lock}_p\rangle \xhookrightarrow{\mathit{lock}_p} \langle q, \mathit{lock}_p\ \mathit{lock}_p\rangle$$
$$\langle q, \mathit{lock}_p\rangle \xhookrightarrow{\mathit{unlock}_p} \langle q, \epsilon\rangle$$

The PDS for m instantiates the above template with the name of each ESL process p, thus defining its transition system. In the rules, lock_p (unlock_p) represents an attempt by process p to lock (unlock) mutex m.

Unit-of-work process As described in §3, violations occur when a Java thread is executing a unit of work. Thus, it is necessary to know if the ESL process that models a Java thread is executing within a unit of work. To track whether an ESL process is executing within a unit of work, `esl2cpds` creates a *unit process* for each ESL process. Like a lock process, the unit process uses the stack to model the status of the ESL process to which it corresponds. That is, it changes its state when it receives a message indicating that a unit of work begins or ends. The following PDS rules define the unit-of-work transition system for an ESL process p:

$$\langle q, \mathit{out}\rangle \xhookrightarrow{p.\mathit{beg}} \langle q, p.\mathit{beg}\ \mathit{not}\rangle \qquad \langle q, p.\mathit{beg}\rangle \xhookrightarrow{p.\mathit{is}} \langle q, p.\mathit{beg}\rangle$$
$$\langle q, p.\mathit{beg}\rangle \xhookrightarrow{p.\mathit{beg}} \langle q, p.\mathit{beg}\ p.\mathit{beg}\rangle \qquad \langle q, \mathit{not}\rangle \xhookrightarrow{p.\mathit{not}} \langle q, \mathit{not}\rangle$$
$$\langle q, p.\mathit{beg}\rangle \xhookrightarrow{p.\mathit{end}} \langle q, \epsilon\rangle$$

Notice that the rule $\langle q, p.\mathit{beg}\rangle \xhookrightarrow{p.\mathit{is}} \langle q, p.\mathit{beg}\rangle$, along with the one labeled with $p.\mathit{not}$, enables other processes to query whether a particular ESL process is currently executing in a unit of work.

ESL processes Each Java thread is modeled by an ESL process. The thread's control flow is modeled by a PDS, which is created following the construction outlined in §4. Additionally, the reads and writes to members of F and U_α are tracked by labeling the PDS rules with an action that describes the ESL statement for which the PDS rule models. For example, in Fig. 1, the field `elementCount` is written to on line 6. This is modeled in the PDS for the Java thread created on line 26 by the following rule: $\langle q, n_6\rangle \xhookrightarrow{p_2.\mathit{write}(\mathit{eD})} \langle q, n_7\rangle$ (where eD stands for `elementData`). This rule models the fact that ESL process p_2 is writing to the `elementData` field of an object allocated at allocation site α (line 29).

Monitor process The monitor process models a particular scenario as defined in [1]. Because each scenario is a regular language, it is easily modeled by a PDS that does not contain *push* or *pop* rules (i.e., as a finite-state machine (FSM)). We illustrate this by showing the translation from Scenario 9 for the program in Fig. 1. Recall that the sequence of reads and writes that lead to a violation of Scenario 9 for the program is: $R_{t1}(v.\mathit{elementCount})$, $W_{t2}(v.\mathit{elementCount})$, $W_{t2}(v.\mathit{elementData})$, $R_{t1}(v.\mathit{elementData})$. Empire translates this sequence of read and write actions into the FSM shown in Fig. 3.

For this particular program, there are only 3 threads. In the FSM, eC stands for the field `v.elementCount` and eD for the field `v.elementData` of the `Vector` class shown in Fig. 1. The label $R_{p1}(eC)$ represents the CPDS action corresponding to a read of the field `elementCount` by ESL process p_1 which models the Java thread t_1. The labels $W_{p2}(eC)$, $W_{p2}(eD)$, and $R_{p1}(eD)$ are defined similarly. (For clarity, the diagram omits the actions associated with the ESL process for the `main` thread because they have no effect in this particular violation.) Because units of work can be nested, each time process p_1 ends a unit of work, the monitor process must determine if p_1 is still executing within a unit of work. This is achieved via the transitions labeled $p_1.end$ to the query states Q_{1-4}. Once the monitor has reached a query state, it checks with the unit-of-work PDS to see if p_1 is still executing within a unit of work. This is done through the actions $p_1.is$ and $p_1.not$. If there exists a sequence of actions that drives the FSM to the final state (E), then Empire has found a (potential) violation of Scenario 9.

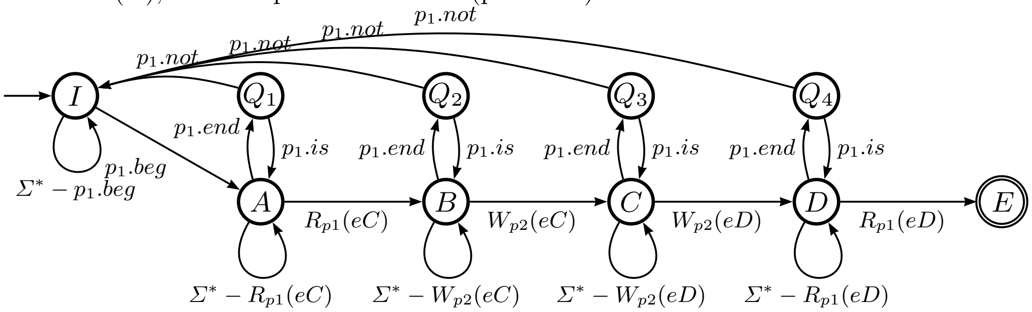

Fig. 3. The FSM that monitors for Scenario 9 [1]. If the CPDS can generate actions that drive the FSM to the accepting state E, then there exists a concurrent execution of the ESL program that violates scenario 9.

6 Experiments

We tested Empire on the programs from the concurrency-testing benchmark created by Eytani et al. [14, 15]. The benchmark suite consists of a set of programs developed by students in an undergraduate software testing class. The student's were instructed to write a concurrent Java program that exhibited a bug. We applied Empire to the programs whose bug was listed as “non-atomic”. All experiments were performed on an Intel(R) Pentium(R) 4 CPU 3.06GHz with 4GB of memory running the Centos operating system with Linux kernel 2.6.9-42. The results are reported in Tab. 2.

Empire reported a counterexample for all but two of the benchmarks tested. In all cases, the counterexample reported was a violation of Scenarios 1, 2, or 4, with most of them being Scenario 1. Most of the time, this was due to a missing `synchronized` statement. As explained in [15], this is typical of the programming errors made by novice programmers.

Qadeer and Wu noticed that many concurrency bugs arise with few interleavings [16] (i.e., they are “shallow” bugs). This is reflected in our results, where the counterexample lengths are typically small.

Program	CG	Total	S	S	Java	Time	Scn	CE	Bug
	Nodes	Insts.	Nodes	Insts.	Thr.	(sec)	#	Len.	
SoftwareVerificationHW	2692	29328	8	173	3	447.9	2	10	F
Test	1017	15747	18	668	3	684.3	1	15	T
BuggedProgram	1012	15206	12	134	3	21.6	2	7	T
bufwriter	1338	19556	11	231	4	OOM	-	-	-
BuggyProgram	1148	17221	8	286	3	37.5	4	9	T
Account	976	14739	7	125	3	30.1	1	8	T
IBM Airlines	1107	16702	6	144	3	22.0	1	6	T
BugTester	1221	18333	20	346	3	20.1	1	6	T
ProducerConsumer	1168	17389	10	471	3	56.3	1	9	T
shop	987	14887	11	249	3	OOM	-	-	-

Table 2. Results of running Empire on the known buggy scenarios in the concurrency testing benchmark. Column CG Nodes reports the total number of call-graph nodes. Column Total Insts. reports the total number of instructions in the program. Column S Nodes reports the number of call-graph nodes after model reduction. Column S Insts. reports the total number of instructions in the reduced model. Column Java Thr. reports the number of Java threads being modeled. Column Time reports the total analysis time (OOM means that the analyzer exahusted memory). Column Scn reports the scenario number for which a counterexample was found. Column CE Len. reports the number of actions in the counterexample. Column Bug reports whether the counterexample was a true bug. “T” signifies that we have verified that the counterexample found was a true violation. “F” signifies that the counterexample was a false positive.

The counterexample returned for the program “SoftwareVerificationHW” is a false positive. This is because we use a sound, but imprecise, modeling of the Java method `Thread.join()`.

In two of the benchmarks, “bufwriter” and “shop”, the CPDS model checker ran out of memory before terminating. In particular, this occurred when the CPDS model checker was intersecting the regular languages that approximate the languages of the PDSs. At present, it is unclear what aspect of these programs lead to this behavior. The two programs are not any larger, in terms of call-graph nodes, than the others. We are currently investigating the nature of the generated regular languages that lead to memory being exhausted.

Tab. 2 shows how Empire behaves when one has perfect insight into what query to issue. To test how Empire behaves when the user has less-than-perfect insight into the program’s behavior, we ran Empire on `Account` for all ESL-process/atomic-set-field combinations for Scenario 1 (with a given allocation site and atomic set). Note that Empire is intended to be used to test *user-generated* hypotheses about possible atomic-set/scenario combinations, so this represents a more realistic usage scenario—i.e., where some amount of insight has been supplied by the user. This involved running reachability queries for 12 CPDSs, which ran in times ranging from 19.7 seconds to 37.2 seconds (with one outlier of 1310.8 seconds); the total time required was 1628 seconds (317.8 seconds without the outlier). Note that each such query is entirely independent from every other query, so they could have been run in parallel on a distributed-computing resource, such as a cluster or a Condor pool.

There are two points to be learned from our experiments. First, they validate the approach we have chosen in the design of Empire. By building a tool that focuses on one static-allocation site at a time, Empire is capable of finding violations in most of the programs analyzed. Second, the experiments show the importance of using model-reduction techniques. In all of the benchmark programs, the size listed in column "S Insts." is less than 5% of the number of instructions in the original program. Because of the state-space explosion problem, model reduction is an important technique when working with software model checkers.

7 Related Work

Several categories of related work can be distinguished: detection of low-level data races, high-level data races, conflict-serializability violations, and atomicity violations. Due to space constraints, we only discuss related work on high-level data-race detection and conflict-serializability.

High-level data-race detection In [6], Artho et al. first coined the term high-level data race. They formalized the notion of *view consistency*, where a *view* expresses what fields are guarded by a lock. Our work is based on atomic-set serializability; see the following discussion that compares atomic-set serializability with view serializability.

Naik and Aiken [17] recently proposed a static analysis for detecting data races based on conditional must-not aliasing. The basic idea is to demonstrate the absence of data races by showing that whenever two locks are different, their guarded locations must be different. Empire does not require this condition to hold.

Vaziri et al. [1] proposed a *data-centric* approach to synchronization. They present an algorithm that automatically infers where to place synchronization when given a program annotated with atomic sets and units of work. Our work is based on their notions, but focuses on checking the properties of programs that already contain occurrences of synchronization primitives.

Conflict Serializability Wang and Stoller [18] present two Commit-Node Algorithms for checking two variants of serializability, view serializability and conflict serializability. In their terminology, two traces are conflict-equivalent iff (i) they contain the same events, and (ii) for each pair of conflicting events, the two events appear in the same order. Two traces are view-equivalent iff (i) they contain the same events, (ii) each read event has the same write-predecessor in both traces, and (iii) each variable has the same trace-final write event in both traces. A trace is serial if the events of each transaction form a contiguous subsequence of the trace, where transaction boundaries are determined by method boundaries, lock acquisition/release, start/join operations of threads, and a few other constructs. A trace is conflict-serializable if it is conflict-equivalent to a serial trace, and a trace is view-serializable if it is view-equivalent to a serial trace. A set of transactions is view-atomic (conflict-atomic) if every trace of it is view-serializable (conflict-serializable).

Our work is based on atomic-set serializability, which is a more liberal criterion than conflict serializability. Atomic-set serializability is also more liberal than view serializability, except that with view serializability two writes by process p_1 interleaved with a write by process p_2 is considered serializable.

Xu et al. [19] present SVD, a dynamic detector of serializability violations. Their work does not require user annotation of atomic regions (called Computation Units or CUs) because it dynamically approximates CUs using a *region hypothesis*. They use dynamic replay and post-mortem analysis to determine if there was a violation. Our work differs in two respects: first, our analysis is a static analysis; second, by allowing the user to specify what object to check, we provide a lightweight approach to checking for atomic-set-serializability violations.

References

1. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL. (2006)
2. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.E.: Eraser: A dynamic data race detector for multithreaded programs. TCS **15**(4) (1997)
3. Engler, D., Ashcraft, K.: RacerX: Effective, static detection of race conditions and deadlocks. In: SOSP. (2003)
4. Pratikakis, P., Foster, J.S., Hicks, M.: LOCKSMITH: Context-sensitive correlation analysis for race detection. In: PLDI. (2006)
5. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: PLDI. (2006)
6. Artho, C., Havelund, K., Biere, A.: High-level data races. In: Proc. NDDL/VVEIS'03. (2003)
7. Hoare, C.: Proof of correctness of data representations. Acta Informatica **1** (1972)
8. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: POPL. (2003)
9. Chaki, S., Clarke, E.M., Kidd, N., Reps, T.W., Touili, T.: Verifying concurrent message-passing C programs with recursive calls. In: TACAS. (2006)
10. Wang, L., Stoller, S.D.: Runtime analysis for atomicity for multi-threaded programs. Technical Report DAR-04-14, State University of New York at Stony Brook (2005)
11. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, TUM (2002)
12. : T. J. Watson Libraries for Analysis (WALA) http://wala.sourceforge.net/wiki/index.php.
13. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Trans. Softw. Eng. Methodol. **14**(1) (2005)
14. Eytani, Y., Ur, S.: Compiling a benchmark of documented multi-threaded bugs. IPDPS **17** (2004) 266a
15. Eytani, Y., Havelund, K., Stoller, S.D., Ur, S.: Towards a framework and a benchmark for testing tools for multi-threaded programs. Concurrency and Computation: Practice and Experience **19**(3) (2007)
16. Qadeer, S., Wu, D.: KISS: Keep it simple and sequential. In: PLDI. (2004)
17. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: POPL. (2007)

18. Wang, L., Stoller, S.D.: Accurate and efficient runtime detection of atomicity errors in concurrent programs. In: PPoPP. (2006)
19. Xu, M., Bodík, R., Hill, M.D.: A serializability violation detector for shared-memory server programs. In: PLDI. (2005)