Generating Data Race Witnesses by an SMT-based Analysis *

Mahmoud Said[1], Chao Wang[2], Zijiang Yang[1], and Karem Sakallah[3]

[1] Department of Computer Science, Western Michigan Univerisity, Kalamazoo, MI 49008
[2] NEC Laboratories America, 4 Independence Way, Suite 200, Princeton, NJ 08540
[3] Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan 48109

Abstract. Data race is one of the most dangerous errors in multi-threaded programming, and despite intensive studies, it remains a notorious cause of failures in concurrent systems. Detecting data races is already a hard problem, and yet it is even harder for a programmer to decide *whether* or *how* a reported data race can appear in the actual program execution. In this paper we propose an algorithm for generating debugging aid information called *witnesses*, which are concrete thread schedules that can deterministically trigger the data races. More specifically, given a concrete execution trace, e.g. non-erroneous one which may have triggered a warning in Eraser-style data race detectors, we use a symbolic analysis based on SMT solvers to search for a data race witness among alternative interleavings of events of that trace. Our symbolic analysis precisely encodes the sequential consistency semantics using a scalable predictive model to ensure that the reported witness is always feasible.

Keywords: Data Race, Debug, SMT, Concurrent Programs

1 Introduction

A data race occurs in a multithreaded program when two threads access the same memory location with no ordering constraints enforced in between, and at least one of the accesses is a write. Programs containing data races are difficult to debug because they may exhibit different behaviors under the same input. In practice, a single synchronization error caused by data race can take weeks for programmers to identify [3, 21]. For the Java Memory Model (JMM) and other relaxed memory models, it is absolutely crucial to remove all data races in user applications even if they do not appear to cause logic errors, because these models guarantee sequential consistency only to race-free programs [15].

Stateful model checking is one of the approaches for finding bugs in concurrent programs [10, 11, 23]. As more scalable exhaustive techniques, statelss model chekers [2, 16] have been developed. Being exhaustive in nature, model

* The work was supported in part by NSF Grants CCF-0811287, CCF-0810865 and ONR Grant N000140910740.

checkers in principle can be used to provide counter-examples. Unfortunately, most existing model checking tools do not scale.

The numerous static and dynamic techniques that have been developed to detect data races [8, 1, 6, 18, 17, 13, 24, 5, 9], except for exhaustive techniques, can only report data race warnings, often in the form of pairs of program locations. None of these methods provide *witnesses* to help the programmers deterministically reproduce the reported data race during actual program executions. By witness, we mean a concrete thread schedule of the program execution that leads to a program state in which two concurrent events with data conflict are both enabled. It is essential debugging information for programmers to decide whether the race is benign, and subsequently figure out how to fix it.

The problem of generating witnesses is orthogonal to detecting data races. The latter problem, which have been studied extensively, ends with a set of *data race warnings*. The witness generation starts from where the data race detection ends, with the goal of providing a concrete thread schedule to reproduce each data race during execution. The witness generation problem is significantly harder, since it has to concern with the feasibility (or existence) of particular concrete executions. It is also a practically important problem with no satisfying solution yet.

In this paper we present an algorithm to generate data race witnesses in multithreaded Java programs based on analyzing a single execution trace. The key idea is to perform a postmortem analysis on a log of the access events. Here we can use any of the existing data race *detection* algorithms [8, 1, 6, 18, 17, 13, 24, 5, 9] to compute a set of *potential* data races, which then act as input to our witness generation algorithm. Given a trace and a set of potential data races, we model the access events of that trace using suitable classes of constraints and formulating the witnesses generation problem as constraint solving. What these constraints represent is not just the given trace itself, but a *maximal set* of interleavings of events of that trace, and all these *alternative* traces are guaranteed to be actual program executions. The constraints generated by our algorithm are in a quantifier-free first-order logic. They can be decided by off-the-shelf Satisfiability Modulo Theory (SMT) solvers, and therefore can benefit from the significant performance advances in recent SMT solvers (e.g. [4]).

Our symbolic predictive model improves over the maximal causal model (MCM) proposed by Serbănută, Chen and Rosu [22]. We improve over the MCM based method in the following aspects. First, the MCM considers semaphores as the only synchronization primitives, whereas in this paper, we precisely model a wide range of synchronization primitives in Java, including wait, notify, and notifyall. Second, the search algorithm used in [22] is based on explicitly enumerating the feasible interleavings, which may become a bottleneck for practical uses; in our method, we conduct the search symbolically using an SMT solver.

To further reduce the overhead of the symbolic search, we pre-simplify the SMT formulas by applying a trace-based conservative analysis [14]. Our analysis is based on computing lock acquisition histories and a must-happen-before relation defined by thread creation/join and matching wait/ notify/notifyall. The goal is to reduce the cost of the more precise, but also expensive, symbolic analysis, by quickly weeding out (bogus) data races that do not have concrete

witnesses. The constraints derived from this analysis can also be added as hints to speed up the SMT search.

We have implemented the proposed method for multithreaded Java programs. Our trace logging is implemented using an agent interface that captures the Java Virtual Machine Execution events, and our symbolic analysis uses the Yices SMT solver [4]. Our preliminary results on public benchmarks show that the witness generation algorithm is scalable enough as a post-mortem analysis, to help programmers better understand the data races.

2 Multithreaded Trace

2.1 Execution Traces

We consider a multithreaded Java program as a set of concurrently running threads, and use $Tid = \{1, \ldots, n\}$ to denote the set of thread indices. The operations on global or *shared* variables are called visible operations, while those on thread-local variables are called invisible operations. In particular, synchronization primitives such as operations on locks and condition variables are regarded as visible operations. An execution trace π is a sequence of instances of *visible* operations in a concrete execution of the multithreaded program. Each instance is called an *event*. For Java programs, both read/write accesses to shared variables and the synchronization operations are recorded as events, while invisible operations are ignored. An event is represented as a tuple $(tid, type, var, val)$, where tid is the thread index, $type$ is the event type, var is either a shared variable (in read/write) or a synchronization object, val is either a concrete value (in read/write) or the child thread index (in thread creation/join). The event type is one of $\{read, write, fork, join, acquire, release, wait, notify, notifyAll\}$. They can be classified into three categories:

1. $read$ and $write$ denote the read and write access to a shared variable, where var is the variable and val is the concrete value;
2. $fork$ and $join$ denote the creation and termination of a child thread, where $(tid, fork, -, val)$ creates a child thread whose index is val, and $(tid, join, -, val)$ joins the child thread back;
3. the rest correspond to synchronization operations over locks and condition variables. The `synchronized` keyword is translated into a pair of $acquire$ and $release$ events over the lock implicitly associated with an object.

For an event e and its attribute a, we will use $e.a$. In addition, given an execution π and an event e in it, $e.idx$ denote the unique index of event e in π. For example, in event $e_i : (1, fork, -, 2)$, we have $e_i.tid = 1, e_i.type = fork, e_i.val = 2$, and $e_i.idx = i$.

2.2 Partial Order and Linearizations

Let $\pi = e_1 \ldots e_n$ be a concrete execution. The trace can be viewed as a total order of the set $\{e_1, \ldots, e_n\}$ of events. To capture all the alternative and yet feasible interleavings of the events in π, we define a *partially ordered set*, denoted $\mathcal{T}_\pi = (T, \sqsubseteq)$, such that

- $T = \{e \mid e \text{ is an event in } T_\pi\}$.
- $\sqsubseteq$ is a partial order such that
  - if $e_i.tid = e_j.tid$ and e_i appears before e_j in π, then $e_i \sqsubseteq e_j$,
  - if $e_i = (tid_1, fork, -, tid_2)$ and e_j is the first event of thread tid_2 in π, then $e_i \sqsubseteq e_j$,
  - if $e_i = (tid_1, join, -, tid_2)$ and e_j is the last event of thread tid_2 in π, then $e_j \sqsubseteq e_i$.
  - $\sqsubseteq$ is transitively closed.

That is, $\mathcal{T}_\pi$ orders events from the same thread based on their execution order in π, but does not order events from different threads except for fork and join.

In the presence of shared variables and synchronization primitives, not all linearizations (total orders) of $\mathcal{T}_\pi$ correspond to actual program executions. We define a *sequentially consistent linearization* τ_π of $\mathcal{T}_\pi$ as one that satisfies $\sqsubseteq$ as well as the following requirements:

- *Write-Read Consistency*: the value read by an event is always written by the most recent write in τ_π, and
- *Synchronization Consistency*: τ_π does not violate the semantics of the synchronization events.

The set of all linearizations of $\mathcal{T}_\pi$ forms the search space of our witness generation algorithm. That is, we search for a sequentially consistent linearization that leads to a state in which two data-conflict events are both enabled.

Our notion of sequentially consistent linearization is inspired by the maximal causal model in [22]. However, the maximal causal model considers semaphore as the only synchronization primitive, and does not explicitly model thread creation and join ($fork$ and $join$), whereas we precisely model a wide range of Java synchronization primitives. Our symbolic method for searching sequentially consistent linearizations is also related to the symbolic predictive analysis [25] based on *concurrent trace programs (CTPs)*. However, in CTPs each event is not a concrete read or write (as in our case) but a symbolic statement derived from the program source code. The concurrent trace program in general captures more feasible interleavings, but it is also more expensive to check.

```
class Value {
1    private int x = 1;
2    public synchronized void add(Value v) {
3        x = x+v.get();
4    }
5    public int get() {
6        return x;
7    }}
class Task extends Thread {
8    Value v1; Value v2;
9    public Task(Value v1, Value v2) {
10       this.v1 = v1;
11       this.v2 = v2;
12   }
13   public void run() {
14       v1.add(v2);
15   }}
class Main {
16   public static void main (String[] args) {
17       Value a = new Value();
18       Value b = new Value();
19       Thread t2 = new Thread (new Task(a, b));
20       Thread t3 = new Thread (new Task(b, a));
21       t2.start();
22       t3.start();
23   }}
```

Fig. 1. A Java program with data races.

As an example, consider the Java program in Figure 1. Inside the `main` method, thread $t1$ creates threads $t2$ and $t3$, which execute methods $t1.run()$

and $t2.run()$, respectively. The shared variables are $a.x$ and $b.x$. Note that, according to the Java execution semantics, $a.x$ is aliased to $t2.v1.x$ and $t3.v2.x$, and $b.x$ is aliased to $t2.v2.x$ and $t3.v1.x$.

Let $Tid = \{1, 2, 3\}$. Executing the program may result in the following partial trace, i.e. a subsequence of events from threads $t2$ and $t3$ as follows: ... (2,13-14), (2,2-3), (2,5-7), (2,4), (2,15), (3,13-14), (3,2-3), (3,5-7), (3,4), (3,15), where each event is denoted as a pair of the thread index and the line number(s). During this execution, the shared variable $b.x$ is read by thread $t2$ at line 6 (aliased as $t2.v1.x$) and written by thread $t3$ at line 3 (aliased as $t3.v2.x$). However, this trace is not a witness of data race because the two aforementioned accesses to $b.x$ are never simultaneously enabled. There exists an alternative interleaving of the same set of events: ... (2,13-14), (2,2-3), (2,5), (3,13-14), (3,2), **(2,6)**, **(3,3)**, (3,5-7), (3,4),(3,15), (2,7), (2,4), (2,15). It is a data race witness because there exists a state in which the read access by event (2,6) and the write access by event (3,3) are both enabled. It is guaranteed to be an actual program execution because both write-read consistency and synchronization consistency

The goal of our symbolic analysis is to search for witnesses among all sequentially consistent linearizations of $\mathcal{T}_\pi$ derived from the concrete execution π. We formulate the data race witness generation problem as a satisfiability problem. That is, we construct a quantifier-free first-order logic formula ψ_π such that the formula is satisfiable if and only if there exists a sequentially consistent linearization of $\mathcal{T}_\pi$ that leads to a state in which two data-conflict events are both enabled. The formula ψ_π is a conjunction of the following subformulas

$$\psi_\pi \ := \ \alpha_\pi \wedge \beta_\pi \wedge \gamma_\pi \wedge \rho_\pi$$

In Section 3 we present algorithms to encode the partial order (α_π), write-read consistency (β_π), and data race property (ρ_π) in first-order logic (FOL) formulas. In Section 4 we discuss the encoding of synchronization consistency (γ_π).

3 Symbolic Encoding of The Write-Read Consistency

3.1 Encoding the Partial Order

Given a multithreaded trace π, let $\pi|_t = \langle e_1^t, \ldots, e_n^t \rangle$ be a sub-sequence that is a projection of π onto the thread t. Let $t.first$ and $t.last$ be the first and last event of thread t in π,i.e., e_1^t and e_n^t, respectively. For each event e, we introduce an event order (EO) variable whose value represents its position in a linearization of $\mathcal{T}_\pi$. To ease our presentation, we assume that an EO variable shares the same unique index with the corresponding event. Therefore $o_{e.idx}$ is the EO variable for e. Let the number of events be $|\pi|$. The domain of o_i, where $1 \leq i \leq |\pi|$, is $[1..|\pi|]$. Furthermore, we have $o_i \neq o_j$ if $i \neq j$.

Equation 1 encodes the partial order requirement of sequentially consistent linearizations of $\mathcal{T}_\pi$. It enforces a total order within each thread-local sequence $\pi|_t (1 \leq t \leq N)$, and enforces the order between the first (or last) event of a thread and the corresponding fork (or join) event, if such event exists. In Equation 1 $FORK$ and $JOIN$ denote the set of $fork$ and $join$ events in $\mathcal{T}_\pi$. For

an event $e \in FORK$, $e.val$ gives the child thread index, thus $(t_{e.val}).first.idx$ is the index of the first event in the child thread.

$$\alpha_\pi \equiv \left(\bigwedge_{t=1}^{T} \left(o_{e_1^t.idx} < \cdots < o_{e_n^t.idx} \right) \wedge \bigwedge_{e \in FORK} \left(o_{e.idx} < o_{(t_{e.val}).first.idx} \right) \wedge \bigwedge_{e \in JOIN} \left(o_{(t_{e.val}).last.idx} < o_{e.idx} \right) \right) \tag{1}$$

$$\beta_\pi \equiv \bigwedge_{e \in \pi \wedge e.type = read} \left(\begin{array}{l} \left((e.tiwp = null) \wedge (e.val = e.var.init) \wedge \bigwedge_{e1 \in e.pws} (o_{e.idx} < o_{e1.idx}) \right) \vee \\ \bigvee_{e1 \in e.pwsv} \left((o_{e1.idx} < o_{e.idx}) \wedge \bigwedge_{e2 \in e.pws \wedge e2 \neq e1} (o_{e.idx} < o_{e2.idx} \vee o_{e2.idx} < o_{e1.idx}) \right) \end{array} \right) \tag{2}$$

$$\rho_\pi \equiv \bigvee_{(e1,e2) \in PDR} ((o_{e1'.idx} < o_{e2.idx} < o_{e1''.idx}) \wedge (o_{e2'.idx} < o_{e1.idx} < o_{e2''.idx})) \tag{3}$$

Figure 2 show an execution trace π with 11 events $e_0, \ldots, e_{10}$ generated by two threads. The last column in Figure 2 lists the partial order constraints: α_1 and α_2 enforces a total order on the events from thread 1 and 2, respectively; α_3 ensures that the fork of thread 2 happens before the first event in thread 2.

$e_0 : (1, fork, -, 2)$ $e_1 : (1, write, x, 1)$ $e_2 : (1, acquire, o, -)$ $e_3 : (1, write, x, 0)$ $e_4 : (1, wait, o, -)$ $e_5 : (2, acquire, o, -)$	$e_6 : (2, read, x, 0)$ $e_7 : (2, notifyAll, o, -)$ $e_8 : (2, release, o, -)$ $e_9 : (2, read, x, 0)$ $e_{10} : (1, release, o, -)$	partial order: $\alpha_1 : o_0 < o_1 < o_2 < o_3 < o_4 < o_{10}$ $\alpha_2 : o_5 < o_6 < o_7 < o_8 < o_9$ $\alpha_3 : o_0 < o_5$ write-read consistency: $\beta : (o_6 < o_1 \vee o_3 < o_6) \wedge (o_9 < o_1 \vee o_3 < o_9)$

Fig. 2. An execution with initial value $x = 0$.

3.2 Encoding Write-Read Consistency

Given a linearization l, we use $e_1 \prec_l e_2$ to denote that event e_1 happens before e_2 in l. Similarly, we use $e_1 \prec_t e_2$ to denote that e_1 happens before e_2 within the same thread t.

Definition 1. Linearization Immediate Write Predecessor*: Given a read event e in a linearization l, we define its* linearization immediate write predecessor *, denoted as $e.liwp$, to be a write event $e' \prec_l e$ such that $e.var = e'.var$ and there does not exist another write event e'' such that $e' \prec_l e'' \prec_l e$ and $e''.var = e.var$.*

Definition 2. Thread Immediate Write Predecessor*: Let $\pi|_t$ be the projection of execution π onto thread t. The* thread immediate write predecessor *to a read event e, denoted as $e.tiwp$, is a write event $e' \prec_t e$ in $\pi|_t$ such that $e.var = e'.var$ and there does not exist another write event e'' such that $e' \prec_t e'' \prec_t e$ and $e''.var = e.var$.*

Definition 3. Write-Read Consistency*: A linearization l is write-read consistent iff for any read event e (1) if there exists a write event e' such that $e' = e.liwp$, then $e.val = e'.val$; (2) if e' does not exist, then $e.val = e.var.init$. Here $e.var.init$ is the initial value of variable $e.var$.*

Definition 4. Predecessor Write Set*: Given an execution π, the* predecessor write set *of a read event e, denoted as $e.pws$ is a set that includes any write event e' such that $e'.var = e.var$ and (1) $e'.tid \neq e.tid$, or (2) $e'.tid = e.tid$ and $e' = e.tiwp$. The* predecessor write of the same value set *to a read event e, denoted as $e.pwsv$, is a subset of $e.pws$, where for any $e' \in e.pwsv$, we have $e'.val = e.val$.*

Equation 2 considers all the possible linearizations that satisfy the write-read consistency requirement. For each read event e in π, there are two possible cases:

1. e has no thread immediate write predecessor ($e.tiwp = null$), its read value is the same as the variable's initial value ($e.val = e.var.init$), and all the write events in the predecessor write set of e happen after e ($o_e.idx < o_{e1}.idx$). Note that the two equality constraints evaluate to either true or false statically, and therefore will not be added in the SMT formula.
2. e follows a write event $e1$ in its predecessor write of the same value set ($o_e.idx < o_{e1}.idx$), and all other writes to $e.var$ happens either before $e1$ ($o_{e2}.idx < o_{e1}.idx$), or after e ($o_e.idx < o_{e2}.idx$). This constraint guarantees that e reads the value written by $e1$ and no other writes can interfere with this write-read pair.

If all the read events satisfy the above constraints, as specified in Equation 2, the linearizations are write-read consistent. Consider the example in Figure 2. Column 3 shows the write-read constraints, along with some implementation optimizations, described as follows:

1. $o_6 < o_1$ requires that the read event e_6 appears before any write to x. Note that although $o_6 < o_3$ is also required as in Equation 2, it is removed (constant true) because it is implied by $(o_6 < o_1)$ together with α_1.
2. $o_3 < o_6$ requires that the read event e_6 happens after e_3. Although the full constraint as in Equation 2 is $(o_3 < o_6) \wedge (o_1 < o_3 \vee o_6 < o_1)$, we remove the second conjunct because $o_1 < o_3$ is implied by α_1.

3.3 Encoding the Data Race

Definition 5. Data Race Witness*:An execution $\pi = \pi_1 e_1 e_2 \pi_2$, where π_1 and π_2 are the trace prefix and suffix, respectively, has a data race on e_1 and e_2 if the two events belong to different threads, access the same shared variable and at least one access is a write.*

Let PDR be the set of potential data races in $\mathcal{T}_\pi$, where each data race is represented as a pair $(e1, e2)$ of events that belong to different thread ($e1.tid \neq e2.tid$), access the same variable ($e1.var = e2.var$), and at least one access is a write ($e1.type = write \vee e2.type = write$).

Given every event pair $(e1, e2) \in PDR$, let $e1'$ and $e1''$ be the events immediately before and after $e1$ in the same thread, and $e2'$ and $e2''$ be the events immediately before and after $e2$ in the same thread. Equation 3 captures the existence of a witness in which $e1$ and $e2$ are simultaneously reachable.

We can further reduce the number of data race constraints (currently 4) into 3 by adding $o_{e1.idx} < o_{e2.idx}$, since it implies the two existing constraints $o_{e1'.idx} < o_{e2.idx}$ and $o_{e1.idx} < o_{e2''.idx}$. A data race exists in an execution π if $e1$ is immediately followed by $e2$ in π. We do not need to consider the dual case that $e1$ immediately follows $e2$ because if such linearization exists, since it is guaranteed that the linearization in which $e2$ follows $e1$ exists as well.

4 Symbolic Encoding of The Synchronization Consistency

4.1 Synchronization Interpretation

The interpretation of the synchronization operations involves replacing object variables with simple-type variables available to SMT solvers, and map the synchronization operations on objects to logic operations on simple-type variables. Although Java allows recursive locks, they happen rarely in executions. An execution π has a recursive lock if there exist two events e_i and e_j in π such that $e_i = e_j = (t, acquire, o, -)$ and there is no event $(t, release, o, -)$ in between; otherwise π is called *recursive-lock-free*. If an execution π is recursive-lock-free, then any sequentially consistent linearization of $\mathcal{T}_\pi$ is also recursive-lock-free (a reorder of events within the same thread is not allowed). In this section we discuss the interpretation for recursive-lock-free executions and defer the discussion for executions with recursive locks until Section 4.3.

We introduce the following simple-type shared variables for each object o.

- An integer variable o_o with domain $[0..N]$, where N is the number of threads. Object o is free if o_o is 0. Otherwise o_o is the thread index that owns object o.
- N Boolean variables $o_{w\_t} (1 \leq t \leq N)$. The value of o_{w_t} is true iff thread t is in object o's wait set.

In the following we list the interpretation of the synchronization operations. For each variable v, we use the normal form v to indicate its current value, and use the primed version v' to indicate its value at the next step.

- Event $(t, acquire, o, -)$ is interpreted as $o_o = 0 \rightarrow o'_o = t$. It requires that the object is free, and then set the owner of object o to thread t.
- Event $(t, release, o, -)$ is interpreted as $o_o = t \rightarrow o'_o = 0$. It requires that the owner of object o is thread t, and then set object o to be free.
- Event $(t, wait, o, -)$ is converted into two consecutive atomic events. The first atomic event is interpreted as $(o_o = t \rightarrow o'_{w\_t} \wedge o_o = 0)$, which requires that the owner of thread o is thread t, and then sets object o to free and the flag o'_{w_t} to true. The second atomic event is interpreted as $(o_o = 0 \wedge \neg o_{w\_t}) \rightarrow o'_o = t$, which requires that object o is free and thread t is no longer waiting. For the $wait$ event to complete, a $notify$ or $notifyAll$ event from another thread needs to interleave in between to reset o_{w_t}.

- Event $(t, notifyAll, o, -)$ is interpreted as $o_o = t \rightarrow \bigwedge_{t1 \in o.wait} \neg o'_{w\_t1}$, where $o.wait$ is the set of threads waiting on object o. It requires that the owner of o is thread t, and then reset o_{w_t1} for any waiting thread $t1$.
- Event $(t, notify, o, -)$ requires that *one and only one* thread waiting on o, if any, is waken up. We introduce N auxiliary variables H_{w_t} with domain $\{0, 1\}$, one for each thread $t \in Tid$, such that (1) H_{w_t} must have value 0 if thread t is not waiting for on o and (2) exactly one H_{w_t} has value 1 if the waiting set for o is not empty. The requirement can be obtained by the following constraints: $\bigwedge_{1 \leq t \leq N}(\neg o_{w\_t} \rightarrow \neg H_{w\_t} = 0)$, $(\bigvee_{1 \leq t \leq N} o_{w\_t}) \rightarrow (\Sigma_{1 \leq t \leq N} H_{w\_t} = 1)$ Finally, the $notify$ event is interpreted as $\bigwedge_{t \in Tid}(H_{w\_t} = 1 \rightarrow \neg o'_{w\_t} \wedge H_{w\_t} = 0 \rightarrow o'_{w\_t} = o_{w\_t})$, which states that thread t is no longer waiting on object o if it is chosen; otherwise its waiting status remains the same.

4.2 The Recursive-lock-free Encoding

Table 1. Recursive-lock-free synchronization consistency Interpretation

Synchronization Event	Interpretation	Predecessor Write Set	Predecessor Write Set with Same Value
$e_2 : (1, acquire, o, -)$	$o_o = 0 \rightarrow o'_o = 1$	$o_o : \{e_5, e_8\}$	$o_o : \{e_8\}$
$e_4 : (1, wait, o, -)$	$o_o = 1 \rightarrow o'_{w\_1} \wedge o'_o = 0$	$o_o : \{e_2, e_5, e_8\}$	$o_o : \{e_2\}$
e'_4	$o_o = 0 \wedge \neg o_{w\_1} \rightarrow o'_o = 1$	$o_o : \{e_4, e_8, e_5\}$ $o_{w\_1} : \{e_4, e_7\}$	$o_o : \{e_4, e_8\}, o_{w\_1} : \{e_7\}$
$e_5 : (2, acquire, o, -)$	$o_o = 0 \rightarrow o'_o = 2$	$o_o : \{e_2, e_4, e'_4, e_{10}\}$	$o_o : \{e_4, e_{10}\}$
$e_7 : (2, notifyAll, o, -)$	$o_o = 2 \rightarrow \neg o'_{w\_1}$	$o_o : \{e_2, e_4, e'_4, e_5, e_{10}\}$	$o_o : \{e_5\}$
$e_8 : (2, release, o, -)$	$o_o = 2 \rightarrow o'_o = 0$	$o_o : \{e_2, e_4, e'_4, e_5, e_{10}\}$	$o_o : \{e_5\}$
$e_{10} : (1, release, o, -)$	$o_o = 1 \rightarrow o'_o = 0$	$o_o : \{e'_4, e_5, e_8\}$	$o_o : \{e'_4\}$

In this section we present the constraints that enforce synchronization consistency for recursive-lock-free multithreaded traces. The first two columns in Table 1 give the interpretation of the synchronization events in Figure 2. The original wait event e_3 is split into two new events: e_3 and and its shadow event e'_3. Correspondingly we introduce an event order variable o'_3 and adds partial order constraint $o_3 < o'_3 < o_4$.

Definition 6. Initial Value*: The initial value $v.iv$, is defined as follows: (1) the value for a variable o_o that denotes the ownership of an object is 0, i.e. $o_o.iv = 0$, (2) the value for a variable that denotes whether thread t is waiting for an object is false, i.e. $o_{w\_t}.iv = false$ for $1 \leq t \leq N$.*
Assumed Value*: The assumed value of a variable v in a synchronization event e in the format of assume $\rightarrow$ update, denoted $v_e.av$, is the value specified in the sub-formula e.assume. Here v is called an* assumed variable *in e, and e.assume is the set of assumed variables in e.*
Written Value*: The written value of a variable v in a synchronization event e in the format of assume $\rightarrow$ update, denoted as $v_e.wv$, is the value specified in the sub-formula e.update. v is called an* updated variable *in e, and e.updated is the set of updated variables in e.*

$$\gamma_e \equiv \bigwedge_{v \in e.assume} \left(\begin{array}{l} \left(v_e.av = v.iv \wedge v_e.first \wedge \bigwedge\limits_{e_1 \in v_e.pws} o_{e.idx} < o_{e_1.idx} \right) \vee \\ \bigvee\limits_{e_1 \in v_e.pwsv} \left(\begin{array}{l} (o_{e.idx} < o_{e_1.idx}) \wedge \\ \bigwedge\limits_{e_2 \in v_e.pws \wedge e_2 \neq e_1} (o_{e.idx} < o_{e_2.idx} \vee o_{e_2.idx} < o_{e_1.idx}) \end{array} \right) \end{array} \right) \quad (4)$$

Given a synchronization event e, Equation 4 enforces a valid position in any linearization for e with respect to other synchronization events. It considers each assumed variable v in e, and adds constraints on the position of e based on the v's assumed value:

- If v's assumed value in e, $v_e.av$, is the same as v's initial value $v.iv$, then e can be in a position that is before any write to v. That is, $\bigwedge\limits_{e_1 \in v_e.pws} o_{e.idx} < o_{e_1.idx}$. Note that if there exist writes to v before e from the same thread, this constraint contradicts the partial order constraint thus becomes false.
- Event e follows an event $e_1 \in v_e.pwsv$. In this case e happens after $e_1 (o_{e_1.idx} < o_{e.idx})$ so the assumed value at e can take updated value at e', and other events that write to v do not interfere by happening either before the write at e_1 or after the read at e.

Column 3 and 4 in Table 1 list the predecessor write set of the shared variables o_o and o_{w_1} and its subset, predecessor write with the same value set, respectively. Table 2 gives the encoding based on Equation 4. Although in Equation 4 there is a constraint $\left(v_e.av = v.iv \wedge \bigwedge\limits_{e_1 \in v_e.pws} o_{e.idx} < o_{e_1.idx} \right)$, the constraint can be removed if v_e's value is not the same as the initial value, or be reduced to $\bigwedge\limits_{e_1 \in v_e.pws} o_{e.idx} < o_{e_1.idx}$ if the values are the same. In addition, several other straightforward optimizations can be applied. Column 3 gives more concise encoding than Column 2 due to the following optimizations:

- A sub-formula s that can be implied by partial order constraint. For example, $o_6 < o_9$ in e_1 and $o_1 < o_3$ in e_3. This reduces $s \wedge s'$ to s, and $s \vee s'$ to $true$.
- A sub-formulas s that contradicts partial order constraint. For example, $o'_3 < o_3$ in e_4 and $o_5 < o_3$ in e_6. This reduces $s \vee s'$ to s.
- A sub-formula s that is weaker than s' in $s \wedge s'$. For example, in $o_1 < o_6 \wedge o_1 < o_9$ in e_1, $o_1 < o_9$ can be removed because $o_6 < o_9$.

Finally the synchronization consistency constraint is specified by $\gamma_\pi \equiv \bigwedge\limits_e \gamma_e$, where e is a synchronization event in π.

4.3 Encoding with Recursive Locks

If an execution π has recursive locks, we define a variable $depth_o^t$ that denotes the depth of object o that has been locked by thread t. The initial value of $depth_o^t$ is 0. For each sequence $\pi|_t$ that is a projection of π on thread t, we increase the value of $depth_o^t$ by 1 for each $(t, acquire, o, -)$, and decrease the value by 1 for each $(t, release, o, -)$. Depending on the value of $depth_o^t$, acquire and release events are encoded differently as the following:

Table 2. Recursive-lock-free synchronization consistency encoding

Event	Encoding	Encoding with Optimization
e_2	$(o_2 < o_5 \wedge o_2 < o_8) \vee ((o_8 < o_2) \wedge (o_5 < o_8 \vee o_2 < o_5))$	$(o_2 < o_5) \vee (o_8 < o_2)$
e_4	$(o_2 < o_4) \wedge (o_5 < o_2 \vee o_4 < o_5) \wedge (o_8 < o_2 \vee o_4 < o_8)$	$(o_5 < o_2 \vee o_4 < o_5) \wedge (o_8 < o_2 \vee o_4 < o_8)$
e'_4	$\begin{pmatrix} (o_4 < o_{4'}) \wedge (o_5 < o_4 \vee o_{4'} < o_5) \\ \wedge (o_8 < o_4 \vee o_{4'} < o_8) \\ (o_8 < o_{4'}) \wedge (o_4 < o_8 \vee o_{4'} < o_4) \\ \wedge (o_5 < o_8 \vee o_{4'} < o_5) \end{pmatrix} \vee$	$\begin{pmatrix} (o_4 < o_{4'}) \wedge (o_5 < o_4 \vee o_{4'} < o_5) \\ \wedge (o_8 < o_4 \vee o_{4'} < o_8) \\ ((o_8 < o_{4'}) \wedge (o_4 < o_8)) \end{pmatrix} \vee$
	$(o_7 < o'_4) \wedge (o_4 < o_7 \vee o'_4 < o_4)$	$(o_7 < o'_4) \wedge (o_4 < o_7)$
e_5	$\begin{pmatrix} (o_5 < o_2 \wedge o_5 < o_4 \wedge o_5 < o'_4 \wedge o_5 < o_{10}) \vee \\ ((o_4 < o_5) \wedge (o_2 < o_4 \vee o_5 < o_2) \wedge \\ (o'_4 < o_4 \vee o_5 < o'_4) \wedge (o_{10} < o_4 \vee o_5 < o_{10})) \vee \\ ((o_{10} < o_5) \wedge (o_2 < o_{10} \vee o_5 < o_2) \wedge \\ (o'_4 < o_{10} \vee o_5 < o'_4) \wedge (o_4 < o_{10} \vee o_5 < o_4)) \end{pmatrix}$	$(o_5 < o_2) \vee (o_{10} < o_5) \vee$ $(o_4 < o_5 \wedge o_5 < o'_4 \wedge o_5 < o_{10})$
e_7	$(o_5 < o_7) \wedge (o_2 < o_5 \vee o_7 < o_2) \wedge (o_4 < o_5 \vee o_7 < o_4) \wedge$ $(o'_4 < o_5 \vee o_7 < o'_4) \wedge (o_{10} < o_5 \vee o_7 < o_{10})$	$(o_2 < o_5 \vee o_7 < o_2) \wedge (o_4 < o_5 \vee o_7 < o_4) \wedge$ $(o'_4 < o_5 \vee o_7 < o'_4) \wedge (o_{10} < o_5 \vee o_7 < o_{10})$
e_8	$(o_5 < o_8) \wedge (o_2 < o_5 \vee o_8 < o_2) \wedge (o_4 < o_5 \vee o_8 < o_4) \wedge$ $(o'_4 < o_5 \vee o_8 < o'_4) \wedge (o_{10} < o_5 \vee o_8 < o_{10})$	$(o_2 < o_5 \vee o_8 < o_2) \wedge (o_4 < o_5 \vee o_8 < o_4) \wedge$ $(o'_4 < o_5 \vee o_8 < o'_4) \wedge (o_{10} < o_5 \vee o_8 < o_{10})$
e_{10}	$(o'_4 < o_{10}) \wedge (o_5 < o'_4 \vee o_{10} < o_5) \wedge (o_8 < o'_4 \vee o_{10} < o_8)$	$(o_5 < o'_4 \vee o_{10} < o_5) \wedge (o_8 < o'_4 \vee o_{10} < o_8)$

- An event $e : (t, acquire, o, -)$ is called the *first acquire event* if $e.depth^t_o = 0$. Its corresponding constraint is $o_o = 0 \rightarrow o'_o = t$.
- For event $e : (t, acquire, o, -)$ that is not a first acquire event, its corresponding constraint is $o_o = t \rightarrow o'_o = t$.
- An event $e : (t, release, o, -)$ is called the *last release event* if $e.depth^t_o = 0$. Its corresponding constraint is $o_o = t \rightarrow o'_o = 0$.
- For event $e : (t, release, o, -)$ that is not a last release event, its corresponding constraint is $o_o = t \rightarrow o'_o = t$.

We do not need to explicitly record the depth of recursive locks. It is based on the observation that (1) π is a valid execution, thus the number of acquire and release events must be balanced; and (2) The depths of recursive locks associated with an acquire or release event (a thread-local property) will not be changed by thread interleavings.

4.4 Correctness and Complexity

Theorem 1. *Let π be the given multithreaded trace. There exists a data race witness in a sequentially consistent linearization of $\mathcal{T}_\pi$ iff ψ_π is satisfiable:*

$$\psi_\pi \equiv \alpha_\pi \wedge \beta_\pi \wedge \gamma_\pi \wedge \rho_\pi$$

According to the definitions of partial order constraint α_π, write-read consistency constraint β_π, and synchronization consistency constraint γ_π, a linearization of $\mathcal{T}_\pi$ that satisfies $\alpha_\pi \wedge \beta_\pi \wedge \gamma_\pi$ is sequentially consistent. Since the events are all from a real execution, a sequentially consistent linearization represents events from a valid execution as well. In addition, the definition of data race property enforces that in the linearization there are two adjacent events (at least one is a write event) from different threads accessing the same variable.

Our approach eliminates the bogus warnings reported by typical data race detection algorithms, e.g. those based on lock-set analysis. Consider the execution shown in Figure 3 where x, y are shared variables with initial value 0. A lock-set analysis will reports a data race warning between the two write events to y as one of them is not protected by any lock. Our approach will not produce a

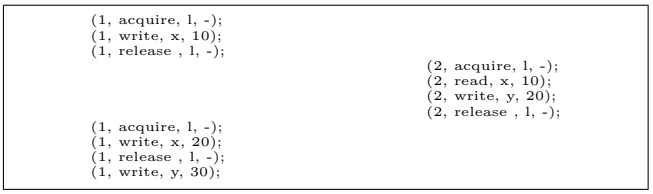

Fig. 3. An execution with shared variables x, y.

data race witness because write-read consistency enforces the read event of x in thread 2 must happen between the two write events to x in thread 1. In addition, each corresponding acquire-release pair is atomic according the synchronization constraints. Therefore the two write events are never enabled at the same time.

For most Java executions the number of synchronization events is very small compared with the number of total events. Since the majority of the constraints are generated from encoding read, write events and data race properties, their complexity determines the scalability of our approach. We note that these constraints are in pure integer difference logic (IDL) – an efficiently decidable subset of FOL where each IDL constraint is of the form $(x - y \leq c)$, where x and y are integer variables and c is 0.

5 Static Optimizations

In the implementation, we use the incremental feature of the Yices SMT solver [4]. We divide the constraints in ψ_π into two parts: $\psi_\pi = (\alpha_\pi \wedge \beta_\pi \wedge \gamma_\pi) \wedge \rho_\pi$, where the first part encodes all the sequentially consistent linearizations, and the second part states that a data race exists. Let ρ_π be a conjunction of subformulas $\rho_\pi(e_i, e_j)$, each of which states the simultaneous reachability of an event pair $(e_i, e_j) \in PDR$. Instead of building and checking ρ_π in one step (same as combining all potential data races in one check), we check each individual event pair in isolation. The incremental SAT procedure is as follows.

1. Within the SMT solver, we first construct the subformula $(\alpha_\pi \wedge \beta_\pi \wedge \gamma_\pi)$.
2. Then for the first data race event pair we construct $\rho_\pi(e_i, e_j)$ and add this subformula as a *retractable* assertion. The retractable assertion can be removed after satisfiability checking, while allowing the SMT solver to retain the lemmas (clauses) learned during the process. If the result is satisfiable, then the SMT solver returns a satisfying assignment (witness); otherwise, such witness does not exists.
3. After retracting the first assertion $\rho_\pi(e_i, e_j)$, we construct $\rho_\pi(e'_i, e'_j)$ for the second event pair (e'_i, e'_j) and add it to the SMT solver.

We keep repeating steps 2 and 3 till all the event pairs in PDR are checked. The benefit of using incremental SAT is reducing the overall runtime by sharing the cost of checking different data races. Although it might appear to be costly to

call the SMT solver once for each potential data race in PDR, the entire process turns out to be efficient because of incremental SAT[1].

Typical data race detection algorithms (e.g. those based on locksets) have false alarms—sometimes many of them, which means the input to our witness generation algorithm, the set PDR of (potential) data races, may have event pair (e_i, e_j) such that e_i, e_j are not simultaneously reachable. Therefore, it is often advantageous to check, before calling the precise SMT analysis, whether (e_i, e_j) simultaneously reachable by using a conservative analysis. Our analysis is based on statically computing the following information: (1) lock acquisition histories [14]; (2) must-happen-before constraints, where event e_1 must happen before e_2 iff that is the case in every linearization of $\mathcal{T}_\pi$. This analysis is in general comparable to and sometimes more precise than standard data race detectors (e.g. [8, 1, 6, 18, 17, 13, 24, 5, 9]).

6 Experiments

We have implemented the proposed method and conducted experiments on some public benchmarks. We collected traces using a Java agent interface that captures the Java Virtual Machine Execution events. Our symbolic analysis is implemented using the Yices SMT solver [4]. All benchmark programs are accompanied by test cases to facilitate the concrete execution. Our experiments were conducted on a workstation with 2.8 GHz processor and 2GB memory.

Table 3. Performance of the symbolic data race witness generation algorithm

Test Program		Given Trace (events)			Shared Variables			Witness Generation			
name	threads	length	lk-evs	wn-evs	rw	lk	wn	lsa	mhb	wtns	time (s)
Example run1	3	25	4	0	6	2	0	8	2	1	0.01
Example run2	3	29	8	0	6	2	0	6	0	0	0.01
Remote Agent	3	45	12	5	6	3	4	12	4	2	0.01
connectionpool	4	85	16	5	5	1	3	21	0	0	0.01
liveness.BugGen	7	241	44	6	12	9	6	138	10	1	0.36
account #1	6	336	82	10	17	11	5	125	45	4	0.09
account #2	11	651	162	20	32	21	10	250	90	9	0.28
account #3	21	1281	322	40	62	41	20	500	180	19	0.79
SyncBench #1	2	107	22	0	3	2	0	8	2	1	0.01
SyncBench #2	13	722	156	0	16	3	0	805	333	40	18.3
BarrierBench #1	7	407	80	14	10	2	7	229	12	0	0.7
BarrierBench #2	13	653	136	28	16	2	7	361	38	0	2.04
philo	6	1050	126	41	23	6	22	563	0	0	0.0
hedc	10	1457	234	0	85	23	0	508	164	40	57.7
Daisy	3	1998	330	14	34	9	12	328	16	7	5.65
elevator	4	8000	1298	0	121	12	0	12	0	0	0.0
tsp	4	45637	20	5	42	5	3	83	4	3	0.05

Table 3 shows the experimental results. Among the benchmarks, `Example (run 1)` is the simple example illustrated in Figure 1, `Example (run 2)` is the same example except that the `get` method is synchronized. All other benchmarks are publicly available in [12, 20, 10, 19, 7]. The first two columns show the statistics of the test program, including the name and the number of threads. The

[1] Often the first few SAT calls take a significant portion of the total runtime; after that, the "learned clauses" make the subsequent SAT calls extremely fast.

next three columns show the statistics of the given trace, including the length (visible events only), the number of acquire/release events, and the number of wait/notify/notifyAll events. The next three columns show the number of data variables (rw), the number of lock variables (lk) and the number of condition variables (wn) in the trace. The last four columns show the statistics of the symbolic witness generation algorithm, including the number of potential data races after the lock acquisition history analysis (lsa), the number of potential data races after the must-happen-before analysis (mhb), the number of witnesses generated (wtns), and the runtime of our symbolic algorithm in seconds. During symbolic witness generation, we call the SMT solver incrementally, one at a time, only for the potential data races in the column *mhb*. The runtime in seconds is the combined processing time for all these potential data races.

The runtime results show that our witness generation algorithm scale to medium length traces, and is fast enough to be used as a postmortem analysis. In almost all cases, our static pruning based on lock acquisition history and must-happen-before constraints is able to reduce the number of potential data races significantly, therefore reducing the burden on the symbolic algorithm. We also note that, even after pruning, most of the potential data races do not have concrete witnesses – they are likely to be bogus errors. This result highlights the problem associated with many data race detection algorithms in the literatures. Reporting such data races (warnings) directly to programmers could be counter-productive in practice, since it imposes significant burden (manual effort) on the programmers for deciding whether a reported data race is real.

7 Conclusion

Despite that numerous static and dynamic techniques exist to detect data races, few are capable of providing witnesses to help programmers understand how a data race can happen during program execution. In this paper we propose a SMT-based symbolic method to produce concrete witnesses for data races in concurrent programs. Our tool can be integrated seamlessly with traditional testing procedure because of the following reasons: (1) the inputs to our tool are ordinary program execution traces, (2) our approach amplifies the effectiveness of each testing run by considering all the alternative event interleavings, (3) the witnesses produced by our tool pinpoint data races and thus help programmers better understanding the erroneous behaviors. Our experimental results show that the proposed algorithm is scalable enough for a postmortem analysis.

References

1. Boyapati, C., Rinard, M.C.: A parameterized type system for race-free Java programs. In: OOPSLA2001. SIGPLAN Notices, vol. 36(11), pp. 56–69. ACM (Nov 2001)
2. Chao Wang, Mahmoud Said, A.G.: Coverage guided systematic concurrency testing. In: International Conference on Software Engineering (ICSE'11) (2011)
3. Christey(editor), S.: Top 25 most dangerous programming errors. CWE/SANS report (2009), `http://cwe.mitre.org/top25/`

4. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Computer Aided Verification. pp. 81–94. Springer-Verlag (2006)
5. Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: a race and transaction-aware Java runtime. j-SIGPLAN 42(6), 245–255 (jun 2007)
6. Engler, D., Ashcraft, K.: RacerX: effective, static detection of race conditions and deadlocks. In: ACM Symposium on Operating Systems Principles. pp. 237–252. ACM (2003)
7. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: Parallel and Distributed Processing. p. 286.2. IEEE Computer Society, Washington, DC, USA (2003)
8. Flanagan, C., Freund, S.: Type-based race detection for Java. In: Programming Language Design and Implementation. pp. 219–232. ACM (2000)
9. Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. In: Programming Language Design and Implementation. pp. 121–133. ACM, New York, NY, USA (2009)
10. Havelund, K.: Using runtime analysis to guide model checking of java programs. In: SPIN. pp. 245–264. Springer-Verlag (2000)
11. Havelund, K., Pressburger, T.: Model checking JAVA programs using JAVA PathFinder. International Journal on Software Tools for Technology Transfer (STTT) 2(4), 366–381 (Mar 2000)
12. http://research.microsoft.com/qadeer/cav issta.htm: Joint cav/issta special even on specification, verification, and testing of concurrent software
13. Kahlon, V., Yang, Y., Sankaranarayanan, S., Gupta, A.: Fast and accurate static data-race detection for concurrent programs. In: Computer Aided Verification. pp. 226–239. Springer (2007)
14. Kahlon, V., Ivancic, F., Gupta, A.: Reasoning about threads communicating via locks. In: CAV. pp. 505–518 (2005)
15. Manson, J., Pugh, W., Adve, S.V.: The java memory model. In: Principles of Programming Languages (2005)
16. Musuvathi, M., Qadeer, S., Ball, T., Musuvathi, M., Qadeer, S., Ball, T.: Chess: A systematic testing tool for concurrent software. Tech. Rep. MSR-TR-2007-149, Microsoft Research (2007)
17. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: Principles of programming languages. ACM (2007)
18. Pratikakis, P., Foster, J., Hicks, M.: LOCKSMITH: context-sensitive correlation analysis for race detection. In: Programming Language Design and Implementation. pp. 320–331. ACM (2006)
19. von Praun, C., Gross, T.R.: Static detection of atomicity violations in object-oriented programs. Object Technology 3(6) (2004)
20. http://www2.epcc.ed.ac.uk/computing/ research_activities/java_grande/index_1.html: The java grande forum benchmark suite
21. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.E.: Eraser: A dynamic data race detector for multi-threaded programs. ACM Trans. Comput. Syst. 15(4), 391–411 (Nov 1997)
22. Serbănută, T.F., Chen, F., Rosu, G.: Maximal causal models for multithreaded systems. Tech. Rep. UIUCDCS-R-2008-3017, University of Illinois at Urbana-Champaign (2008)
23. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Using model checking with symbolic execution to verify parallel numerical programs. In: ISSTA (2006)
24. Voung, J., Jhala, R., Lerner, S.: RELAY: static race detection on millions of lines of code. In: Foundations of Software Engineering. pp. 205–214. ACM (2007)
25. Wang, C., Kundu, S., Ganai, M., Gupta, A.: Symbolic predictive analysis for concurrent programs. In: International Symposium on Formal Methods. ACM (2009)