

An Operational Semantics including “Volatile” for Safe Concurrency*

John Boyland

University of Wisconsin–Milwaukee, USA
boyland@cs.uwm.edu

Abstract. In this paper, we define a novel “write-key” operational semantics for a kernel language with fork-join parallelism, synchronization and “volatile” fields. We prove that programs that never suffer write-key errors are exactly those that are “data race free” and also those that are “correctly synchronized” in the Java memory model. This 3-way equivalence is proved using Twelf.

1 Introduction

This work is motivated by the desire to define a type system for a Java-like language to prevent data races. Data races are intrinsically a multi-threaded issue. However a scalable type system or program analysis analyzes each thread, indeed every method body separately, using invariants and annotations to ensure that interactions follow desired patterns. It is well known that deadlock can be prevented by requiring that mutexes be acquired in strictly increasing order. Here we show how we can characterize programs without data races in a similar way, that is without explicitly needing to refer to multiple threads.

There are different understandings of what a data race is. At an intuitive level, a data race occurs when an execution of a multi-threaded program leads to the point where two conflicting accesses in two different threads occur “at the same time.” Two accesses are *conflicting* if they are to the same object’s field and one of them is a write. Somewhat more precisely, the current Java memory model (JMM) [15] defines a “happens before” partial order; a program is *correctly synchronized* if in all sequentially consistent executions, two conflicting accesses are always ordered by “happens before.” Reading and writing of “volatile” fields affect the “happens before” order and thus whether a program is correctly synchronized. Both of these techniques explicitly involve reasoning about multiple threads at once.

This paper addresses this situation with the following contributions:

- It defines a simple imperative language with Java-style (re-entrant) monitors, volatile fields and fork-join parallelism. A novel aspect of the operational

* Work supported in part by the National Science Foundation (CCF-0702635). The opinions expressed here are not necessarily those of the National Science Foundation or the US Government.

semantics is that the system constructs and passes “write keys” to simulate the “happens before” relation.

- The paper defines that a program is data-race free if no execution has a “write-key error” in which a thread attempts to access a (non-volatile) field for which it does not possess the write key.
- It is proved that this characterization is equivalent *both* to the intuitive concept of lack of “simultaneous” conflicting accesses, *and* to the JMM-inspired definition of “happens before”-ordered accesses. The proof is mechanically checked in Twelf’s metalogic [17].

A corollary to the last contribution is that the two earlier conceptions of data-race freedom are equivalent, a result that I have not seen previously.

The advantage of the “write-key error” conception for data races is that write-key errors are detected (and can be prevented) thread locally. In other words, if a type system can ensure for each thread that it always possesses the write keys for the (non-volatile) fields that it accesses and that it has exclusive access to fields it writes, then the entire program is thread safe. Moreover, since a write-key error causes a thread to get stuck in our semantics, and since (full) deadlocks always means the program as a whole is stuck, then if a type system enjoys “progress” and “preservation” over the operational semantics, then *per force* the type system will also prevent race conditions.

2 Background on the Current Java Memory Model

This section briefly describes multi-threading primitives in Java and the “happens before” relation of the current Java Memory Model.

In Java, a new thread can be started which executes a given `run` method; we call this a `fork` action. At the other end, one may wait for a thread t to complete execution by executing `t.join()`; this is a `join` action. Thread mutual exclusion is effected by “synchronizing” on an object o : `synchronized (o) { body }`. The runtime system ensures that two separate threads that both synchronize on the same object (known by its role as a *mutex*) mutually exclude each other’s “body” instructions. When a synchronized block is executed, it first attempts to **acquire** the mutex, blocking if some other thread is currently executing a synchronized block on the same object. Once acquisition is successful, the body is executed after which the mutex is **released**. Synchronized statements in Java are “re-entrant” in that if a synchronization block is nested dynamically within another synchronization block on the same object, the inner synchronization succeeds immediately.

Fields in Java may be declared as “volatile.” This designation may be seen as a declaration that these fields will be read and written without mutual exclusion. More importantly, accesses to volatile fields (denoted `readv` and `writv`) constrain the memory model.

A memory model is a contract between the programmer on the one hand and the compiler and the runtime system on the other hand. The most informative model for programmers is a “sequentially consistent” model that indicates that

execution will always be consistent with a system in which the thread interleaving of instructions ensures that each instruction fully executes before the next starts. Sequential consistency however is very limiting for a compiler. Consider the following situation where two threads are executing in parallel, `x` and `y` represent shared mutable locations and `r1` represent thread local “registers.”:

$$\frac{\text{Initially } x = y = 0}{\begin{array}{c|c} x = 1 & r2 = y \\ y = 2 & r1 = x \end{array}}$$

In a sequentially consistent execution, no interleaving of the threads could result in `r1 = 0`, `r2 = 2`. Thus a compiler would not be permitted to reorder the two write actions even though they have no data dependencies.

For such reasons, most memory models only guarantee sequential consistency for fields declared “volatile.” For other fields, threads must use mutual exclusion techniques. The (intuitive) guarantee for normal (non-volatile) fields is that if a program is *data-race free*, that is, if no sequentially consistent execution ever exhibits a “race condition” for a normal field, then that program will enjoy sequentially consistent semantics. A *race condition* is when one thread is ready to write an object’s field when another thread is ready to read or write the same object’s field. If a program *could* exhibit a race condition under a sequentially consistent semantics, then most memory models usually do not guarantee a sequentially consistent semantics. In the small example given above, there is a race condition. Thus a compiler is justified if it wishes to reorder the statements, even though this reordering violates sequential consistency.

In the current Java memory model (JMM) [15], the guarantee is expressed in a different way. First, a “synchronizes with” relation is defined:

1. A `release` action synchronizes with an `acquire` action on the same object;
2. A `writew` action synchronizes with a `readv` action on the same object’s field;
3. A `fork` action synchronizes with the first action in the spawned thread;
4. The last action in a thread synchronizes with a `join` action on that thread.

Additionally the default initialization of a field (with `null` for reference types) synchronizes with all actions in all threads.

Then the “happens before” partial order is defined as the transitive closure of (1) the intra-thread execution order and (2) the “synchronizes with” relation over actions already ordered by the execution.

Specifically of interest to the present paper, the JMM defines what it means to be “correctly synchronized”:

A program is *correctly synchronized* if and only if in all sequentially consistent executions, all conflicting accesses to non-volatile variables are ordered by “happens-before” edges.

The JMM (and generalizations [18]) guarantees that a correctly synchronized program will observe sequentially consistent semantics. This guarantee appears rather different than that expressed concerning “data race free” but as proved in Sect. 5, the definitions are equivalent for our small concurrent language.

```

class Node {
  Node next;
  Node(Node n) { next = n; }

  Node getNext() { next; }

  int count() {
    if this == null then 0
    else 1 + next.count();
  }

  Node copy() {
    if this == null then null
    else new Node(next.copy());
  }

  Node nap(Node n) {
    if this == null then n
    else (next = next.nap(n);
         this);
  }

  void add1() {
    this.nap(new Node(null));
  }
}

```

Fig. 1. A simple node class.

```

class Race {
  Node nodes;

  Race() { }

  int get() {
    nodes.count();
  }

  void inc() {
    nodes = nodes.add1();
  }
}

class Main {
  void main() {
    let t = new Race() in
    ( fork { t.inc(); t.get(); }
      t.inc(); t.get() );
  }
}

```

Fig. 2. A class with an unprotected field; and a test harness.

3 Example

Figure 1 declares a node class. The surface syntax resembles Java, but method bodies contain expressions, not statements. For instance, `getNext()` returns the next field. The `count` method shows another difference: since the language omits dynamic dispatch for simplicity, one can call methods on null references. In the body, one may test for null. In this way, we can model so-called “static” methods. The `copy` method performs a deep copy; `nap` does a destructive append; `add1` extends the list by one node. The `Node` class has mutable state and thus cannot be safely used in a concurrent program without additional restrictions.

We now define several different classes wrapping a node list with the same interface: an `inc` method that adds to the list and a `get` method that counts the size of the current list. The first implementation, `Race` (Fig. 2), does nothing to protect the list. The main program forks off a thread that calls `inc` and `get` and then proceeds to do the same calls in its own thread. Lacking synchronization, the call `t.inc()` in one thread conflicts with `t.inc()` or `t.get()` in the other.

The traditional technique (“standard practice”) for protecting mutable state is to designate a *protecting* object for each piece of mutable state (one object may protect many others) and ensure that all accesses to the state occur dynamically

```

class Traditional {
  Node nodes;

  int get() {
    synch (this) do
      nodes.count();
  }

  void inc() {
    synch (this) do
      nodes = nodes.add1();
  }
}

```

Fig. 3. Traditional approach.

```

class UsingVolatile {
  volatile Node nodes;

  int get() {
    nodes.count();
  }

  void inc() {
    synch (this) do
      nodes = nodes.copy().add1();
  }
}

```

Fig. 4. Using volatility.

only within a synchronization on the protecting object. For example, see class `Traditional` in Fig. 3; the bodies of the methods `get()` and `inc()` include synchronizations around the access of the mutable state.

If `get()` calls are frequent and updates very infrequent, one can do better with a less-known pattern using volatile variables. Figure 4 shows how a volatile field can substitute for synchronization. The reading method can simply access the nodes directly using a volatile field read, and then traverse the list without synchronization. The incrementing method copies the structure before modifying it, to avoid interfering with `get` calls. Furthermore, `inc` is synchronized to ensure that two increments are not carried out in parallel (to preserve “atomicity” [9], an important concept beyond the scope of this paper). We permit interleaving of `get()` and `inc()` calls since the `inc()` method never updates state the `get()` method can see, except for the volatile field.

4 Operational Semantics

This section defines the syntax and dynamic semantics of the paper’s kernel concurrent language. The set of all fields is F . A subset $F_V \subseteq F$ are “volatile” and one $(\text{Lock} \in F_V)$ holds the state of the mutex associated with each object.

4.1 Syntax

Figure 5 gives the syntax. For simplicity, we omit primitive types and arithmetic operators. Expressions include literal object references (natural numbers) and uses of local variables. A new object can be allocated with the given set of fields. Fields of objects can be read or written. The “let,” “if” and “while” constructs are conventional. Procedure calls are included, but not dynamic dispatch because the details would obscure the emphasis of this work.

The concurrency-related terms are fork-join terms (`fork` creates a new thread and starts it; and `join` waits for it to terminate) and synchronization (`synch`

$e ::=$	<i>expression term:</i>	$c ::=$	<i>conditional term:</i>
o	<i>literal reference</i>	true	<i>true</i>
x	<i>program variable</i>	not c	<i>negation</i>
new (\bar{f})	<i>allocation</i>	c and c	<i>conjunction</i>
$e.f$	<i>field read</i>	$e == e$	<i>equality</i>
$e.f := e$	<i>field write</i>	false	\triangleq not true
let $x=e$ in e	<i>local</i>	c or c'	\triangleq
if c then e else e	<i>conditional</i>	not(not c and not $c')$	
while c do e	<i>loop</i>		
$m(\bar{e})$	<i>procedure call</i>		
fork e	<i>fork a thread</i>		
join e	<i>get thread result</i>		
synch e do e	<i>synchronization</i>	$t ::= e \mid c$	<i>term</i>
hold o do e	<i>... in execution</i>	$d ::= m(\bar{x}) = e$	<i>procedure definition</i>
$e_1; e_2$	\triangleq let $_ = e_1$ in e_2	$g ::= d; \dots; d$	<i>program</i>

Fig. 5. Syntax.

and hold). A hold expression is used to indicate that this thread is currently executing a **synch** statement.

The examples in the previous section use a surface syntax with classes, methods and types. A simple translation (not shown) can strip out these features.

4.2 Semantics

This section defines the small-step operational semantics. Novel here is the use of “write keys.” Write keys allow us to separate the notion of “happens before” from considering the execution of multiple threads and instead look at a single thread at a time. A (possibly new) *write key* (a natural number) is generated whenever a normal field is written. This field is added to the *knowledge* of the thread that performed the write. Knowledge is monotonically non-decreasing. Write keys are passed from one thread to another during synchronization actions *indirectly* through memory. In this way, if a write in one thread happens before the read in the other thread, the read is guaranteed to have the necessary key.

The main evaluation relation $(\mu; \theta; \kappa) \xrightarrow{g} (\mu'; \theta'; \kappa')$ relates triples:

μ maps a location $(o.f)$ to a pair (W, o') of a set of write keys and a value. For a normal field, $W = \{w\}$, where w is the key from the most recent write; for a volatile field, W is the set of keys from all threads having written it.

θ maps a thread identifier (natural number) to the expression that the thread is currently executing.

κ maps a thread identifier to its set of known write keys.

g lists the procedure definitions.

The following notation is used for map update:

$$f[x \mapsto v](x') = \begin{cases} v & \text{if } x = x' \\ f(x) & \text{otherwise} \end{cases} \quad f[x \overset{\circ}{\mapsto} v] = f[x \mapsto f(x) \circ v]$$

$$\begin{aligned} \mathsf{T}[\bullet] ::= & \bullet \mid \mathsf{T}[\bullet].f \mid \mathsf{T}[\bullet].f := e \mid o.f := \mathsf{T}[\bullet] \mid m(\bar{o}, \mathsf{T}[\bullet], \bar{e}) \\ & \mid \mathsf{let} \ x = \mathsf{T}[\bullet] \ \mathsf{in} \ e \mid \mathsf{if} \ \mathsf{T}[\bullet] \ \mathsf{then} \ e \ \mathsf{else} \ e \mid \mathsf{synch} \ \mathsf{T}[\bullet] \ \mathsf{do} \ e \\ & \mid \mathsf{hold} \ o \ \mathsf{do} \ \mathsf{T}[\bullet] \mid \mathsf{T}[\bullet] \ \mathsf{and} \ c \mid \mathsf{not} \ \mathsf{T}[\bullet] \mid \mathsf{T}[\bullet] == e \mid o == \mathsf{T}[\bullet] \end{aligned}$$

$$\begin{array}{c} \text{EVAL} \\ \frac{\theta p = \mathsf{T}[t] \quad (\mu; \theta; \kappa; t) \xrightarrow{p}_g (\mu'; \theta'; \kappa'; t')}{(\mu; \theta; \kappa) \xrightarrow{g} (\mu'; \theta' [p \mapsto \mathsf{T}[t']]; \kappa')} \quad \text{E-CALL} \\ \frac{g = \dots; m(\bar{x}) = e; \dots \quad |\bar{x}| = |\bar{o}|}{(\mu; \theta; \kappa; m(\bar{o})) \xrightarrow{p}_g (\mu; \theta; \kappa; [\bar{x} \mapsto \bar{o}]e)} \\ \\ \text{E-LET} \\ (\mu; \theta; \kappa; \mathsf{let} \ x = o_1 \ \mathsf{in} \ e_2) \xrightarrow{p}_g (\mu; \theta; \kappa; [x \mapsto o_1]e_2) \\ \\ \text{E-IF} \\ (\mu; \theta; \kappa; \mathsf{if} \ c \ \mathsf{then} \ e_{\mathsf{true}} \ \mathsf{else} \ e_{\mathsf{false}}) \xrightarrow{p}_g (\mu; \theta; \kappa; e_c) \\ \\ \text{E-WHILE} \\ (\mu; \theta; \kappa; \mathsf{while} \ c \ \mathsf{do} \ e) \xrightarrow{p}_g (\mu; \theta; \kappa; \mathsf{if} \ c \ \mathsf{then} \ e; \ \mathsf{while} \ c \ \mathsf{do} \ e \ \mathsf{else} \ 0) \\ \\ \text{E-NOTNOTTRUE} \quad \text{E-ANDTRUE} \\ \frac{}{(\mu; \theta; \kappa; \mathsf{not} \ \mathsf{false}) \xrightarrow{p}_g (\mu; \theta; \kappa; \mathsf{true})} \quad \frac{}{(\mu; \theta; \kappa; \mathsf{true} \ \mathsf{and} \ c) \xrightarrow{p}_g (\mu; \theta; \kappa; c)} \\ \\ \text{E-ANDFALSE} \\ \frac{}{(\mu; \theta; \kappa; \mathsf{false} \ \mathsf{and} \ c) \xrightarrow{p}_g (\mu; \theta; \kappa; \mathsf{false})} \\ \\ \text{E-EQUALTRUE} \quad \text{E-EQUALFALSE} \\ \frac{o = o'}{(\mu; \theta; \kappa; o == o') \xrightarrow{p}_g (\mu; \theta; \kappa; \mathsf{true})} \quad \frac{o \neq o'}{(\mu; \theta; \kappa; o == o') \xrightarrow{p}_g (\mu; \theta; \kappa; \mathsf{false})} \end{array}$$

Fig. 6. Non-concurrency-related evaluation rules.

Evaluation proceeds using EVAL: a thread is chosen non-deterministically and evaluates one step. Here $(\mu; \theta; \kappa; t) \xrightarrow{p}_g (\mu'; \theta'; \kappa'; t')$ states that thread p in program g makes progress by converting t into t' while side-effecting μ , θ and κ .

For explanatory reasons, the evaluation rules are presented in two groups. The first group of evaluation rules (Fig. 6) are those that have no side-effects. A procedure call uses rule E-CALL once all the parameters are evaluated: we find a procedure in the program with the correct number of arguments and replace the call with the procedure body, substituting the parameters. A **let**-bound variable is substituted in the body once its value is ready. An **if** with a constant boolean is evaluated by choosing the appropriate branch. A **while** loop is converted immediately into an **if**. Conditions use short-circuit evaluation (E-ANDFALSE).

Figure 7 includes the remaining evaluation rules. As mentioned previously, normal fields are associated with a write key that indicates what knowledge is needed to read the field. When a **new** expression is encountered, all of the fields are initialized with null using a write key (0) that all threads know. (This

$$\begin{array}{c}
\text{E-NEW} \\
\frac{f_0 = \text{Lock} \quad (o, \text{Lock}) \notin \text{Dom}(\mu) \quad f_i \text{ distinct}}{(\mu; \theta; \kappa; \mathbf{new} (f_1, \dots, f_n)) \xrightarrow[g]{p} (\mu[(o, f_i) \mapsto (\{0\}, 0) \mid 0 \leq i \leq n]; \theta; \kappa; o)}
\end{array}$$

$$\begin{array}{c}
\text{E-READ} \\
\frac{\mu(o.f) = (\{w\}, o') \quad w \in \kappa(p) \quad f \notin F_V}{(\mu; \theta; \kappa; o.f) \xrightarrow[g]{p} (\mu; \theta; \kappa; o')}
\end{array}$$

$$\begin{array}{c}
\text{E-WRITE} \\
\frac{\mu(o.f) = (\{w\}, -) \quad w \in \kappa(p) \quad f \notin F_V \quad w' \text{ arbitrary} \quad \mu' = \mu[o.f \mapsto (\{w'\}, o')] \quad \kappa' = \kappa[p \overset{\cup}{\mapsto} \{w'\}]}{(\mu; \theta; \kappa; o.f := o') \xrightarrow[g]{p} (\mu'; \theta; \kappa'; o')}
\end{array}$$

$$\begin{array}{c}
\text{E-READV} \\
\frac{\mu(o.f) = (W, o') \quad \text{Lock} \neq f \in F_V \quad \kappa' = \kappa[p \overset{\cup}{\mapsto} W]}{(\mu; \theta; \kappa; o.f) \xrightarrow[g]{p} (\mu; \theta; \kappa'; o')}
\end{array}$$

$$\begin{array}{c}
\text{E-WRITEV} \\
\frac{\mu(o.f) = (W, -) \quad \text{Lock} \neq f \in F_V \quad \mu' = \mu[o.f \mapsto (W \cup \kappa(p), o')]}{(\mu; \theta; \kappa; o.f := o') \xrightarrow[g]{p} (\mu'; \theta; \kappa; o')}
\end{array}$$

$$\begin{array}{c}
\text{E-FORK} \\
\frac{(p, \text{Lock}) \notin \text{Dom}(\mu)}{(\mu; \theta; \kappa; \mathbf{fork} e) \xrightarrow[g]{p} (\mu[(p', \text{Lock}) \mapsto (\{0\}, 0)]; \theta[p' \mapsto e]; \kappa[p' \mapsto \kappa(p)]; p')}
\end{array}$$

$$\begin{array}{c}
\text{E-JOIN} \\
\frac{\theta(p') = o}{(\mu; \theta; \kappa; \mathbf{join} p') \xrightarrow[g]{p} (\mu; \theta; \kappa[p \overset{\cup}{\mapsto} \kappa(p')]; o)}
\end{array}$$

$$\begin{array}{c}
\text{E-RE-ENTER} \\
\frac{\mu(o.\text{Lock}) = (\emptyset, p)}{(\mu; \theta; \kappa; \mathbf{synch} o \mathbf{do} e) \xrightarrow[g]{p} (\mu; \theta; \kappa; e)}
\end{array}$$

$$\begin{array}{c}
\text{E-ACQUIRE} \\
\frac{\mu(o.\text{Lock}) = (W, 0) \quad W \neq \emptyset \quad \mu' = \mu[(o.\text{Lock}) \mapsto (\emptyset, p)] \quad \kappa' = \kappa[p \overset{\cup}{\mapsto} W]}{(\mu; \theta; \kappa; \mathbf{synch} o \mathbf{do} e) \xrightarrow[g]{p} (\mu'; \theta; \kappa'; \mathbf{hold} o \mathbf{do} e)}
\end{array}$$

$$\begin{array}{c}
\text{E-RELEASE} \\
\frac{\mu(o.\text{Lock}) = (\emptyset, p) \quad \mu' = \mu[(o.\text{Lock}) \mapsto (\kappa(p), 0)]}{(\mu; \theta; \kappa; \mathbf{hold} o \mathbf{do} o') \xrightarrow[g]{p} (\mu'; \theta; \kappa; o')}
\end{array}$$

Fig. 7. Remaining evaluation rules.

follows the JMM—default initialization synchronizes with the first action in every thread.) Every object is allocated with a mutex (special field `Lock`).

Field reads and writes of non-volatile fields (E-READ, E-WRITE) require that the thread has knowledge of the write that produced the value: $w \in \kappa(p)$. For a write, an arbitrary write key w' is used to label the new write. In general, this may be one that no thread is aware of. Using such a key would cause the `Race` program in earlier Fig. 2 to get stuck when the second increment executes.

One way in which knowledge of writes can be transmitted is through volatile fields (E-READV, E-WRITEV). Writing a volatile field adds the thread’s knowledge κp to the memory with the written value. When the volatile field is read, the reading thread picks up this knowledge. This follows the JMM rule that says that writing a volatile field synchronizes with all following reads.

For E-FORK, the new thread gets the knowledge of the “forker.” This corresponds to the JMM rule that a fork synchronizes with the first action in the new thread. An object is allocated to represent the thread. In E-JOIN, this thread

can only progress if the other thread has finished execution (down to a value). It gets a copy of all the thread’s knowledge. This follows from the JMM’s rule that the final action in a thread synchronizes with a thread that “joins” it.

During synchronization, the lock’s value is replaced with the number of the acquiring thread, and the knowledge is replaced by the empty set. In E-RE-ENTER, if we synchronize on a lock that this thread already has acquired, the body is simply executed without any effect on the lock. This last rule corresponds to Java’s re-entrant monitors; here, we avoid the need to count multiple entrances because the evaluation rule drops the release action as well as the acquire action.

If the lock is not held by any thread (E-ACQUIRE), the lock field is assigned the number of this thread, and we get the keys from the lock. The synchronization block is then converted into a hold block. When the body has finished evaluation (E-RELEASE), the lock is given the knowledge of the current thread. This knowledge is thus made available for the next thread which acquires the lock. These rules again follow from the JMM.

The semantics defined here is sequentially consistent, but if a thread lacks the necessary write key, it gets stuck. Thus if the program has race conditions, it *may* get stuck (but may not, for instance if an old key is chosen by E-WRITE). A type system for this language that enjoys progress and preservation for *all* executions will prevent this (and the other problems not mentioned). We have designed a type system [?] based on fractional permissions [5, 6] that we believe will achieve this goal, but space precludes including it here.

5 Equivalence

Programs that execute in our operational semantics without ever blocking because of missing write keys are “correctly synchronized” according to (our variant) of the Java Memory Model *and* to the traditional definition of “race free.” In other words, we show a three-way equivalence.

In order to prove equivalence, we need to formally define the aspects we are showing equivalent. To start with, we restrict programs so that they do not include arbitrary object reference constants or partially executed synchronizations:

Definition 1. *A program g is valid if every declaration $m(\bar{x}) = e$ in g has no instance of `hold` nor any literal object reference except the null reference 0.*

Execution starts by calling the `main` procedure in thread 0, which starts with no knowledge except write key 0.

Definition 2. *The initial state I is the state*

$$I = ((0, Lock) \mapsto (\{0\}, 0))[0 \mapsto \text{main}()][0 \mapsto \{0\}] \quad .$$

We formalize what it means for there to be a *write key error* in a program:

Definition 3. *A program $g = \bar{d}$ has a write key error if for some execution $I \xrightarrow[g]{*} (\mu, \theta, \kappa)$ in which a read or write access on a non-volatile field $o.f$ is ready to execute in thread p ($\theta_p = T[o.f := o']$ or $\theta_p = T[o.f]$, where $f \notin F_V$), and the thread does not have the required write key: $(\mu(o.f) = (\{w\}, -)$ and $w \notin \kappa_p)$.*

Definition 4. Two terms t_1 and t_2 are conflicting accesses of a non-volatile field $o.f$ ($f \notin F_v$) if one of them is a write to this field ($t_i = o.f := o'$) and the other is a write ($t_{3-i} = o.f := o''$) or a read ($t_{3-i} = o.f$) of the same field.

Next, we formalize what it means to have a “race condition”: a write access to a field happens at the “same time” as a read access to the same field, and that field is not volatile:

Definition 5. A program $g = \bar{d}; e_0$ exhibits a race condition if there is some execution $I \xrightarrow[g]{*} (\mu, \theta, \kappa)$ such that for two threads $p_1 \neq p_2$ we have $\theta(p_i) = T(t_i)$ and t_1, t_2 are conflicting accesses.

Before we can define what it means to be “correctly synchronized,” we must define an “action” and the “happens-before” relation for actions:

Definition 6. An action λ is an evaluation $(\mu; \theta; \kappa; t) \xrightarrow[g]{p} (\mu'; \theta'; \kappa'; t')$. An evaluation sequence $I \xrightarrow[g]{*} (\mu, \theta, \kappa)$ induces the actions above the line for each instance of EVAL: $\lambda_1, \lambda_2, \dots, \lambda_n$.

Definition 7. Given an execution $\lambda_1, \dots, \lambda_n$, we define a happens-before (written $i \sqsubset j$) relation on the subset of natural numbers $\{1, \dots, n\}$. It is smallest transitive relation that includes the following pairs:

1. $i \sqsubset j$ if $i < j$ and λ_i is an instance of E-RELEASE and λ_j is an instance of E-ACQUIRE on the same object.
2. $i \sqsubset j$ if $i < j$ and λ_i is an instance of E-WRITEV and λ_j is an instance of E-READV on the same field.
3. $i \sqsubset j$ if $i < j$ and $\lambda_i = - \xrightarrow[g]{p} -$ and $\lambda_j = - \xrightarrow[g]{p} -$.
4. $i \sqsubset j$ if $i < j$ and $\lambda_i = (\mu; \theta; \kappa; \text{fork } t) \xrightarrow[g]{p} (\mu'; \theta'; \kappa'; q)$ and $\lambda_j = - \xrightarrow[g]{q} -$.
5. $i \sqsubset j$ if $i < j$ and $\lambda_i = - \xrightarrow[g]{q} -$ and $\lambda_j = (\mu; \theta; \kappa; \text{join } q) \xrightarrow[g]{p} (\mu'; \theta'; \kappa'; t)$.

It can be easily shown that \sqsubset is a partial order compatible with $<$.

Our final definition is for *correctly synchronized* in the style of the JMM:

Definition 8. A program $g = \bar{d}; e$ is correctly synchronized if for any execution of g : $\lambda_1, \dots, \lambda_n$ and any i for which λ_i is an instance of E-WRITE and any j for which λ_j is an instance of E-READ or E-WRITE for the same field, then either $i \sqsubset j$ or $j \sqsubset i$.

It might seem that because our operational semantics detects race conditions, the conflicting access would never execute and thus could not demonstrate an incorrect synchronization, but because write keys are arbitrary, the write could use 0 and thus enable execution. The semantics does not ensure that *all* executions of a program with race conditions will get stuck, just that there will be *some* execution that does.

We now show the three-way equivalence between the three conceptions of race-freedom:

Theorem 1. *The following statements about a valid program g are equivalent:*

1. g exhibits a race condition;
2. g has a write key error;
3. g is incorrectly synchronized.

Proof. (Sketch)

(1) \Rightarrow (2): Suppose we have a program with a race condition. Starting with the execution state that exhibits the race condition, we choose to evaluate the write first. If this write cannot execute because of a missing write key, we are done. Otherwise we choose a new write key not known by the other thread, and we now have a write key error.

(2) \Rightarrow (3): We prove the contrapositive: if the program is correctly synchronized, there will be no write-key error. This is because if there is a happens-before connection between two actions, the thread knowledge of the second will include that produced by the first, and thus the second access will succeed. The connection between write key knowledge and happens-before follows from the fact that the knowledge never decreases (the first case for happens-before) and the other cases for happens-before involve the reader/acquirer getting all the write keys left by the writer.

(3) \Rightarrow (1): Suppose we have an incorrectly synchronized program, in which the code of the first action λ_i is a write executed in thread p and the second action λ_j is an access executed in thread q . (The case that λ_i is a read and λ_j is a write is analogous.)

If the actions are already consecutive, we have the required race condition in the state just before the first executed. Otherwise, we consider how evaluation actions can be reordered (between different threads, never within a thread) to get the accesses adjacent. We partition the intervening actions into those that happen before j and those which do not. The second must include action i , from the definition of incorrect synchronization. We find the last action λ_* in the second group. It cannot be “happens before” any in the first group, or a transitive happens-before relation would exist putting it *in* the first group. Now we reorder it step-by-step with all later actions until it is after λ_j . If λ_* was λ_i , then the last reordering would have resulted in the required race condition. Otherwise, now that it is after λ_j we have reduced the number of intervening instructions. This process must terminate at some point.

6 Extensions

Extending the simple language here to full Java is almost entirely just a matter of complex but uninteresting details. Static fields and static synchronization can be modeled using instance fields and instance synchronization of singleton objects. Types, primitive values and dynamic dispatch have no effect on concurrency. Java 5 adds a new library of synchronization primitives, but it has a reference implementation in core Java. Timeout and timing issues can be modeled by claiming that each step of execution takes 1 nanosecond.

The `Thread` class includes a number of deprecated methods that permit one thread to suspend or terminate another. These we can omit from the formalism. Other methods such as `holdsLock` (because one can only use it to query the current thread) can be implemented without affecting the proof substantially.

Java’s `wait/notify` system would require substantive changes to the formalism. When a thread calls `wait`, it first releases the object’s lock, then it waits to be “notified” and then it waits to re-acquire the lock. The lock release and acquisition lead to the corresponding standard happens-before relations. Another missing piece is thread interruption (and the corresponding interrupted exception). My guess is that the proof could be modified to handle `wait` and interruption.

7 Related Work

The current Java memory model is much more complex than what is modeled here. In particular it gives semantics for programs that are *not* properly synchronized. Since its publication, it has been generalized [18], Apsinall and Ševčík [1] formally prove the main guarantee—that correctly synchronized programs will have a sequentially consistent semantics (whereas the work described here *assumes* sequential consistency). The initialization of reference fields causes some concern which we avoid by using a universally known write key for initialization.

Cenciarelli, Knapp and Sibilio [8] give a vastly different semantics of the Java Memory Model based on configuration structures. As with the papers just reviewed, it handles all Java programs, not just properly synchronized ones, and does not assume sequential consistency. The present author must confess that he was unable to understand the details.

Type systems have been proposed that prevent race conditions and sometimes deadlocks in concurrent programming languages. Flanagan and Abadi [10, 11] define two separate type systems for avoiding races, both of which are accompanied by operational semantics. One is based on Gordon and Hankin’s concurrent object calculus [13] in which mutable objects are represented in the syntax as concurrent processes. The other uses a conventional store. Neither semantics directly detects race conditions, nor includes “volatile.” In either case, a race condition is defined as the (global) possibility that a write could occur at the same time as a read of the same field. (In one system [11], two “simultaneous” reads are also considered a race.) The type system maintains certain invariants that are shown to prevent data races.

Later work (such as Flanagan and Freund [12], Greenhouse [14] and Boyapati and Rinard [4, 3]) omit formal specification of operational semantics, implicitly following the same approach just outlined. Volatile fields, if they are handled at all, are simply regarded as loopholes in the type system.

Permandla and Boyapati [16] define a small-step semantics for a subset of Java virtual machine language (JVML) including synchronization (but not volatile fields) and show that well-typed programs are free of concurrency errors. The semantics however enforces an ownership model and uses method an-

notations that indicate required locking state. The operational semantics is not independent of the type system.

Guava [2] uses a type system to prevent races in a dialect of Java. Guava permits reader/reader parallelism, but omits volatiles. Guava is defined by (informally described) compilation to Java byte-code. Guava is intended as a practical programming language rather than as a minimal concurrent language.

Brookes [7] gives the semantics of a concurrent program by defining its set of “action traces.” Roughly this means that all possible interleavings are considered. A race condition in which a write to mutable state is directly interleaved with another access to the same state is “catastrophic,” in that this particular trace immediately aborts. The semantics omits “volatile.”

8 Conclusions

This paper defines an operational semantics of volatile fields that enables a type system to reason compositionally about them. It uses write keys to detect threading violations. It shows that write-key errors occur if and only if the program may exhibit a race condition, if and only if it is not correctly synchronized.

Acknowledgments

I thank Aaron Greenhouse and Jonathan Aldrich for providing helpful feedback on early drafts. I thank Yang Zhao, Bill Retert and Mohamed ElBendary for frequent conversations on the topic. I thank the many anonymous reviewers for their comments.

SDG

References

1. David Aspinall and Jaroslav Ševčík. Formalising Java’s data race free guarantee. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*. Springer, 1007.
2. David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of Java without data races. In *OOPSLA’00 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications, ACM SIGPLAN Notices*, 35(10):382–400, October 2000.
3. Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. Ph.D., Massachusetts Institute of Technology, February 2004.
4. Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *OOPSLA’01 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications, ACM SIGPLAN Notices*, 36(11):56–69, November 2001.
5. John Boyland. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

6. John Boyland and William Retert. Connecting effects and uniqueness with adoption. In *Conference Record of POPL 2005: the 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 283–295. 2005.
7. Stephen Brookes. A semantics for concurrent separation logic. In *CONCUR 2004 — 15th International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 16–34. Springer, 2004.
8. Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In Rocco De Nicola, editor, *ESOP'07 — Programming Languages and Systems, 16th European Symposium on Programming*, Braga, Portugal, March 24–April 1, volume 4421 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, New York, 2007.
9. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, *ACM SIGPLAN Notices*, 38:338–349, May 2003.
10. Cormac Flanagan and Martín Abadi. Object types against races. In *CONCUR '99—10th International Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 1999.
11. Cormac Flanagan and Martín Abadi. Types for safe locking. In *ESOP'99 — Programming Languages and Systems, 8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108. Springer, 1999.
12. Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, *ACM SIGPLAN Notices*, 35(5):219–232, May 2000.
13. Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In *HCLC '98, 3rd International Workshop on High-Level Concurrent Languages*. Elsevier, September 1998. Published as *Electronic Notes in Theoretical Computer Science* 16 No. 3 (1998), <http://www.elsevier.nl/local/entcs/volume16.html>.
14. Aaron Greenhouse. *A Programmer-Oriented Approach to Safe Concurrency*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2003.
15. Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Conference Record of POPL 2005: the 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 378–391. ACM Press, 2005.
16. Pratibha Permandla and Chandrasekhar Boyapati. A type system for preventing data races and deadlocks in the Java virtual machine language. In *ACM SIGPLAN/SIGBED 2007 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, pages 1–10. ACM Press, June 2007.
17. Frank Pfenning and Carsten Schürmann. Twelf user's guide, version 1.4. Available at <http://www.cs.cm.edu/~twelf>, 2002.
18. Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. A theory of memory models. In *PPoPP'07: ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 161–172. March 2007.
19. Yang Zhao. *Concurrency Analysis Based on Fractional Permissions*. PhD thesis, University of Wisconsin-Milwaukee, 2007.

Auxiliary Materials

See <http://www.cs.uwm.edu/~boyland/papers/simple-concur.html> for how to download the Twelf proof.