IBM Research Report

Associating Synchronization Constraints with Data in an Object-Oriented Language

Mandana Vaziri, Frank Tip, Julian Dolby
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
Associating Synchronization Constraints with Data in an Object-Oriented Language *

Mandana Vaziri  Frank Tip  Julian Dolby
IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA
{mvaziri,ftip,dolby}@us.ibm.com

Abstract
Concurrency-related bugs may happen when multiple threads access shared data and interleave in ways that do not correspond to any sequential execution. Their absence is not guaranteed by the traditional notion of “data race” freedom. We present a new definition of data races in terms of 11 problematic interleaving scenarios, and prove that it is complete by showing that any execution not exhibiting these scenarios is serializable for a chosen set of locations. Our definition subsumes the traditional definition of a data race as well as high-level data races such as stale-value errors and inconsistent views. We also propose a language feature called atomic sets of locations, which lets programmers specify the existence of consistency properties between fields in objects, without specifying the properties themselves. We use static analysis to automatically infer those points in the code where synchronization is needed to avoid data races under our new definition. An important benefit of this approach is that, in general, far fewer annotations are required than is the case with existing approaches such as synchronized blocks or atomic sections. Our implementation successfully inferred the appropriate synchronization for a significant subset of Java’s Standard Collections framework.

1. Introduction
Writing correct concurrent programs is hard, because inconsistent results may be computed when two threads access shared data concurrently. In particular, a data race is said to occur when two threads concurrently access some data, where one of these accesses is a write, and where no synchronization exists between the threads. Current techniques for preventing data races involve obtaining locks prior to any access to the shared data using mechanisms such as Java’s synchronized blocks, or using language constructs such as atomic sections [11] and transactional memory [2, 24, 23] that ensure that a sequence of statements is executed atomically.

One disadvantage of such code-centric approaches for avoiding data races is that it involves non-local reasoning: Shared data may be accessed throughout the program and data races may occur if the programmer forgets to obtain the appropriate locks at any of these points. A second problem is that, even if every access to shared data is protected, data may still end up in an inconsistent state. This situation—sometimes referred to as “high-level data races” [3]—occurs if a consistency property exists between multiple pieces of shared data, and if the synchronization constructs do not ensure that this property is maintained at all times. Avoiding such high-level data races requires the same kind of non-local reasoning as for ordinary data races, but is further complicated by the fact that multiple locks may have to be acquired in a specific order. If the programmer accidentally fails to obey this locking discipline, deadlock or inconsistent data may result.

This paper presents an alternative, data-centric approach for avoiding both high-level and low-level data races. In this approach, the programmer specifies that a consistency property exists between a given set of fields, but without specifying the property itself. We will call such a set an atomic set of fields, indicating that the elements of such a set must be updated atomically. Accesses to fields in an atomic set are assumed to take place in a unit of work, which indicates a logical operation on shared data, and preserves consistency when executed sequentially. In this paper, units of work are assumed to coincide with method bodies. Choosing portions of object state as atomic sets and methods as units of work exploits the encapsulation mechanism of objects.

Given a pair of fields that occur in an atomic set, we have identified 11 problematic interleaving scenarios that capture the various ways in which inconsistent data may occur when two threads update these fields non-atomically. The problematic interleaving scenarios include traditional data races, stale-value errors [10], inconsistent views of data [3], and several other forms of high-level data races. We prove this list of problematic interleaving scenarios to be complete, in the sense that if an execution does not display any of these scenarios, then its projection on each atomic set is serializable, i.e., equivalent to an execution in which the units of work occur in a serial order.

We also present an interprocedural static analysis that determines, for a given atomic set of fields, the places in the code where synchronization must be performed in order to ensure that there are no data races under our new definition. This is implemented by inserting reader-writer locks from the java.util.concurrent library of Java 1.5 [31] at the appropriate places. We implemented the analysis and conducted experiments with classes from the Java Standard Collections Framework. Our experiments indicate that our data-centric approach is sufficient to infer the correct synchronization in a significant portion of the collections framework. Furthermore, one of our constructs can effectively replace synchronization wrappers such as Collections.synchronizedList(). The experiments indicate that the number of atomic location sets is generally far smaller than the number of synchronized blocks, hence reducing the burden on the programmer and creating fewer

* This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.
opportunities for errors. In summary, this paper make the following contributions:

- A list of problematic interleaving scenarios that subsumes the traditional notion of a data race as well as stale-value errors, inconsistent views and other high-level data races. We prove this list to be complete, in the sense that if a program execution does not exhibit these scenarios, then its projection onto each atomic set is serializable.
- A set of data-centric language constructs that allow the programmer to express synchronization constraints succinctly and declaratively.
- A static analysis that infers automatically where synchronization needs to be performed. This relieves the programmer from the non-local reasoning and cumbersome locking disciplines associated with current code-centric approaches.
- Experiments on the Java Standard Collection Framework that illustrate the practicality of the work.

2. Motivating Examples

This section gives some examples that illustrate the shortcomings of the traditional, code-centric approaches for avoiding data races. At the same time, we will introduce the language constructs that are part of the data-centric approach we propose.

2.1 Example 1: Customers

Figure 1(a) shows a class `Customer`, which contains fields `city` and `zipcode` that store parts of a customer’s address, and `date` and `item` that record the item and date of his last purchase. Methods `updateAddress()` and `newPurchase()` serve to update customer information. `PreferredCustomer` is a subclass of `Customer` that models certain aspects of a customer loyalty program using a method `newStoreGift()` that also updates `date` and `item`.

If methods such as `updateAddress()`, `newPurchase()`, and `newStoreGift()` are executed concurrently by multiple threads, care must be taken to ensure that no inconsistent results can arise. For the purposes of this example, we will assume that low-level data races involving any of the four fields are undesirable (e.g., we want to preclude situations where one thread reads the value of `date` while another thread is updating `date` simultaneously). In addition, we want to disallow high-level data races involving the related fields `city` and `zipcode` and involving `date` and `item`. For example, we want to disallow the situation where one thread intends to read `first` `city` and then `zipcode`, but where a second thread writes a new value into `city` before the first thread has completed both reads. In the example of Figure 1(a), atomic sections are used to prevent these low-level and high-level data races.

Conceptually, each atomic section is executed without interruptions by other threads. Atomic sections can be implemented using locks [11] or using transactional memory [2, 24, 23]. The use of atomic sections for preventing data races has the following drawbacks:

- In general, the number of atomic sections may be proportional to the number of accesses to shared fields. In the above example, each method contains an atomic section because it accesses shared data.
- There is a lack of modularity in the sense that the burden is placed on the programmer to remember that accesses to fields in superclasses may have to be protected.

Figure 1(b) shows the approach we propose, in which synchronization constraints are associated with data. Here, all the programmer needs to do is indicate that `city` and `zipcode` are part of an atomic set called `address`, and that `date` and `item` are part of an atomic set called `purchase`. In this framework, the compiler infers where locks must be obtained so as to prevent low-level and high-level data races. Observe that the number of annotations is proportional to the number of fields, and that no additional work is required in the presence of subclassing, thus reducing the amount of work and limiting opportunities for programmer errors.

Informally, the semantics of atomic sets can be stated as follows. Associated with each atomic set `A` is a set of code blocks that represent logical operations on the set. We will refer to these code blocks as the `units of work` for `A`, denoted by `Units(A)`. By default, the units of work associated with an atomic set declared in class `C` consist of the methods of `C` and its subclasses (we will shortly discuss a mechanism for associating additional units of work with a given atomic set). For a given atomic set `A` and unit of work `u` ∈ `Units(A)`, the guarantee is that any pair of accesses to fields in `A` that occur in `u` will be executed without being interleaved by another thread that operates on fields in `A`. For example, methods `Customer.newPurchase()` and `PreferredCustomer.newStoreGift()` are units of work for atomic set `purchase`. Therefore, it is guaranteed that the execution of these methods will not be interleaved, thus preventing high-level data races. However, it is allowed for the execution of one could also use explicit locking mechanisms such as Java’s synchronized blocks to prevent the low-level and high-level data races in this example.
of either of these methods to be interleaved with that of method
\texttt{Customer.updateAddress()}, because the latter does not operate
on the same atomic set.

### 2.2 Example 2: Vector

The default units of work for a given atomic set are well-suited to
accommodate situations where some consistency property between
a set of fields must be maintained by the methods of the class that
delays those fields. However, there are situations where additional
synchronization on parameters is needed.

Figure 2(a) shows a fragment of code \texttt{java.util.Vector}
from the Java Standard Collections Framework. Specifically, the
figure shows the declaration of a field \texttt{elementData}, which refers
to the array that stores the vector’s contents, and a field
\texttt{elementCount}, which counts the number of array elements that
are currently in use. Also shown is a constructor for creating a
new Vector that is initialized to contain the elements of a given
collection \texttt{c}. Wang and Stoller [35] reported a high-level data race
that occurs in this code when this constructor is invoked with a
collection of length \texttt{k}. The race occurs if a thread that executes
the constructor’s code is interrupted after executing the statement
\texttt{elementCount = c.size();} by another thread that is calling the
\texttt{removeAllElements()} method on the collection pointed to by
\texttt{c}. Then, when the first thread resumes, and executes the statement
\texttt{c.toArray(elementData)}, the resulting vector will contain \texttt{k}
elements that are null. This result is inconsistent with any serial
execution of the two threads.

Figure 2(b) shows how this high-level data race can be avoided
using our new language constructs. The fields \texttt{elementCount} and
\texttt{elementData} have been placed in an atomic set \texttt{vec}, and the
constructor has been designated as a unit of work for its parameter
\texttt{c}. The \texttt{unitfor} construct used in this example is a mechanism
for specifying client-side synchronization constraints, and declares
that the scope of parameter \texttt{c} is a unit of work for all atomic sets of
\texttt{c}. Hence, the body of the constructor is not only a unit of work for
all the atomic sets of this but also for those of \texttt{c}.

When a unit of work is declared on multiple atomic sets, as
is the case here, the atomic sets are combined to form a larger
atomic set for the duration of that unit of work. The guarantee is
that accesses to any location within that enlarged set will not be
interleaved. Similarly, a method that accesses fields belonging to
multiple atomic sets of the receiver object is a unit of work for the
union of these sets.

### 2.3 Example 3: Bank Accounts

Figure 3 shows an example program containing classes \texttt{Account}
and \texttt{Bank}. \texttt{Account} has a field \texttt{checking()} and\texttt{deposit()} that manipulate this field. The checking field
has been placed in a singleton atomic set account to prevent
low-level data races involving this field. \texttt{Bank} provides a method
\texttt{transfer()} for transferring money between accounts, and
declares fields \texttt{log} and \texttt{logCount} for maintaining a log of completed
transfers. Observe that \texttt{log} and \texttt{logCount} have been placed in
an atomic set \texttt{logging} to prevent other threads from observing
intermediate states in which only one of the two has been updated.

To make the example slightly more interesting, we will as-
sume that a distinction needs to be made between local trans-
sfers, for which intermediate states (in which the money has been
withdrawn from one account, but not yet added to the other)
should not be visible, and long-distance transfers, for which the
exposure of intermediate states can be tolerated. This distinc-
tion has been encoded by two methods, \texttt{localTransfer()} and
\texttt{longDistanceTransfer()}, both of which invoke the previously
discussed \texttt{transfer()} method. In essence, we would like to ex-
press that \texttt{localTransfer()} is a unit of work for its parameters
\begin{verbatim}
public class Account {
    atomic(account) int checking;
    public void deposit(int n) { } ...
    public void withdraw(int n) { ... }
}
public class Bank {
    atomic(logging) Log log;
    atomic(logging) int logCount;
    void transfer(Account a, Account b, int n) {
        log.add(a, b, n);
        a.withdraw(n);
        b.deposit(n);
        logCount++;
    }
    public void localTransfer(unitfor Account a, unitfor Account b, int n) {
        transfer(a, b, n);
    }
    public void longDistanceTransfer(Account a, Account b, int n) {
        transfer(a, b, n);
    }
}

Figure 3. Bank account example.
\end{verbatim}
a and \texttt{b}, and this is accomplished using the \texttt{unitfor} construct.
As \texttt{localTransfer()} reads both \texttt{a} and \texttt{b}, synchronization will
be inserted to ensure that the call to \texttt{transfer()} will be executed
atomically.

Observe that this solution allows for more concurrency than a
traditional solution where the body of the \texttt{transfer()} method has
been placed in an atomic section in order to preserve the logging
information. This is illustrated by Figures 4 and 5, which show where
calls \texttt{a’.deposit(a) and log’.add(c, d, n)} can be interleaved with
calls to \texttt{localTransfer()} and \texttt{longDistanceTransfer()},
respectively. Note that calls to \texttt{deposit()} can be interleaved with
calls to \texttt{longDistanceTransfer()} while preserving the consis-
tency of \texttt{log} and \texttt{logCount}.

### 2.4 Example 4: Synchronization Wrappers

The Java Collections Framework provides \texttt{synchronization
wrappers} for creating synchronized versions of collections that are not
thread-safe. For example, class \texttt{java.util.ArrayList} provides
array-based lists that are not thread-safe. An application that wishes
to use a thread-safe \texttt{ArrayList} typically executes code such as:
\begin{verbatim}
List myList = Collections.synchronizedList(new ArrayList());
\end{verbatim}

Here, the \texttt{synchronizedList()} method from the utility class
\texttt{java.util.Collections} creates a decorator object of type \texttt{List}
that wraps the \texttt{ArrayList} that was passed in as a parameter,
and that forwards all methods to this \texttt{ArrayList}. All forwarding
methods are synchronized, thus preventing low-level data races
that might otherwise be caused by concurrent accesses to methods
such as \texttt{get()} and \texttt{set()}. Note that this only prevents races when
using a single synchronized wrapper; it is possible to have races if
other threads have references to underlying collection object.

We present an alternative to synchronization wrappers—\textit{atomic}
class—which addresses these shortcomings. In essence, making a
class atomic is equivalent to putting all of its fields and the fields
in its superclasses in a single atomic set. In addition, anonymous
atomic classes can be created by inserting the keyword \texttt{atomic}
public class Vector {
    Object[] elementData;
    int elementCount;

    public Vector(Collection<? extends E> c) {
        elementCount = c.size();
        // 10% for growth
        elementData = new Object[
                (int)Math.min((elementCount*110L)/100,
                Integer.MAX_VALUE)];
        c.toArray(elementData);
    }
}

class LinkedList {
    owned(entry) atomic(list) Entry header;
    atomic(list) int size;
    public set(int index, Object element) {
        Entry e = entry(index);
        oldVal = e.element;
        e.element = element;
        return oldVal;
    }
}

class Entry {
    atomic(entry) Object value;
    owned(entry) atomic(entry) Entry next;
}

Figure 6. Linked List Example

Figure 7. A linked list. Only the objects in the representation of linked list are contained in an atomic set, not the objects contained in the list (shaded).

placement in the list (using a method entry(), not shown) and inserts a given object at that position. We need to declare that all the objects that are part of the representation of the linked list are part of the same atomic set, to protect the entire list, especially in methods such as set() which accesses the list in the middle. To achieve this, we apply the construct owned(entry) to the header field, which states that the entry set of the object pointed to by that field is to be included in the atomic set of that field. This is illustrated in Figure 7. The set list owns the entry set pointed to by field header, which includes the value and next field of the first object. This set in turn includes the entry set of the next object, and so on. Observe that the state of objects pointed to by the value field are not included because this field does not have an owned annotation. Hence, updates to objects in lists can happen concurrently with operations on the list itself.

While this construct could, in general, be expensive to implement with explicit locks, an ownership type system can be leveraged to provide an efficient implementation since owned—that is, private—state does not need additional locks. Integrating our scheme with ownership types is part of future work.
3. New Definition of Data Races

The current definition of a data race is two accesses to the same memory location, one of which is a write with no synchronization between them. This is not sufficient in that the absence of races does not imply the absence of concurrency-related bugs, i.e., bugs caused solely by interleavings of otherwise-correct code. Our objective in providing a new definition for data races is to bridge the gap between traditional data races and a property related to serializability.

Our definition is given as a set of non-serializable interleaving scenarios in Section 3.2. If an execution does not display any of these scenarios, then it satisfies a property related to serializability. We refer to this fact as completeness of the definition, and prove it in Section 3.3.

3.1 Formal Model

This section presents a dynamic formal model of code in terms of sequences of accesses to memory locations, atomic sets, and units of work.

Let $L$ be the set of all memory locations. A subset $L \subseteq L$ may be designated as atomic. An event is an access to a memory location $l \in L$. Accesses can be a read $R(l)$ or a write $W(l)$. We assume that accesses to a single memory location are uninterrupted. If $l$ denotes locations $l_1$ or $l_2$ in $L$, we use the notation $L \setminus l$ to denote the other location.

A unit of work $u$ is a sequence of events, and is declared on a set of atomic sets. We write $\text{sets}(u)$ for the set of atomic sets corresponding to $u$. We say that $\bigcup_{l \in \text{sets}(u)} L$ is the dynamic atomic set of $u$. Units of work may be nested, and we write $u \leftarrow u'$ to indicate that $u'$ is nested in $u$. Units of work form a forest via the $\leftarrow$ relation.

An access to a location $l \in L$ appearing in unit of work $u$ belongs to the top-most unit of work within $u$ such that $L \subseteq \text{sets}(u)$. The notation $R_u(l)$ denotes a read belonging to $u$, and similarly for writes.

As an illustration consider again the example of Figure 3. Figure 8 shows the accesses and units of work in localTransfer(). Unit of work $u$ contains three nested units $u_1$, $u_2$, and $u_3$, corresponding to calls to methods add(), withdraw(), and deposit(), respectively. Unit of work $u_2$ is declared on atomic sets logging of this, account of $a$, and account of $b$, and preserves the consistency of their union. All accesses in this sequence must be protected inside $u$, and we say that all these accesses belong to $u$. This illustrates how, in general, an access belongs to the topmost unit of work declared on it.

A thread is a sequence of units of work. The notation $\text{thread}(u)$ denotes the thread corresponding to $u$. An execution is a sequence of events from one or more threads. Given an execution $E$ and an atomic set $L$, the projection of $E$ on $L$ is an execution that has every event on $L$ in $E$ in the same order.

Observation 1 For any pair of accesses belonging to units of work $u$ and $u'$ that appear in the projection of an execution $E$ on an atomic set $L$, if $\text{thread}(u) = \text{thread}(u')$ then we have neither $u \leftarrow u'$, nor $u' \leftarrow u$; i.e., the projection does not contain nested units of work.

This observation follows from the fact that each access belongs to the topmost unit of work declared on it.

An interleaving scenario is also a sequence of events. For example, $R_u(l) W_{u'}(l) W_v(l)$ is an interleaving scenario where unit of work $u$ first reads $l$, then another unit of work $u'$ performs a write, followed by a write by $v$. An execution is in accordance with an interleaving scenario if it contains the events in the interleaving scenario, and these appear in the same order. The atomic sets of an execution $E$, atomicSets($E$), consists of all atomic sets for which there is an access in $E$, as well as the dynamic atomic set of all units of work in $E$. When the execution is clear from context, we write atomicSets.$$

3.2 Definition

Figure 9 shows the interleaving scenarios that are non-serializable. Serializability is obtained by preserving data consistency, so these scenarios capture when the data may be read or written inconsistently.

Definition 1. Data Races Let $L$ be an atomic set of locations, $l_1, l_2 \in L$, $l$ one of $l_1$ or $l_2$, and $u$ and $u'$ two units of work for $L$, such that $\text{thread}(u) \neq \text{thread}(u')$. An execution has a data race if it is in accordance with one of the interleaving scenarios of Figure 9.

We now describe informally why these scenarios are problematic. In the first scenario, unit of work $u$ reads one location $l$, followed by an update to $l$. If another update to $l$ is interleaved between the two, then the read operation yields a stale value and the subsequent update may be inconsistent. This scenario captures common “low-level” data races, such as two threads executing xx++. Scenario 1 corresponds roughly to the “lost update” [37] anomaly in databases: a transaction $T_1$ reads a data item, then another transaction $T_2$ updates the item, then $T_1$ updates the item based on the value read and commits. The update of $T_2$ is then lost.

Scenario 2 shows two consecutive reads of location $l$ in a unit of work that do not yield the same value. It roughly corresponds...
to the “fuzzy read” anomaly in databases, where a transaction \(T_1\) reads a data item, then a second transaction \(T_2\) modifies that item and commits. If \(T_1\) attempts to re-read the same item, it receives a different value.

In scenario 3, an intermediate value of \(l\) is read, when a unit of work writes it multiple times. In scenario 4, the value read for \(l\) is not the same as the one last written in the same unit of work. In scenario 5, a write to \(l\) is lost, or hidden by the writes from some other unit of work.

Scenarios 6 (Figure 10) and 7 illustrate cases where memory is updated inconsistently. Recall that \(l\) denotes one of \(l_1\) or \(l_2\), and that \(L - l\) denotes the other. In scenario 6, a unit of work updates some location in the set, followed by an update to another location. Thus the whole set is updated in multiple steps. If a write to the set is interleaved between the two, then memory is left in an inconsistent state since individual locations have values from different operations. A reader may then observe what appears to be intermediate states of various updates. Scenario 7 is similar.

Scenarios 8 through 11 (Figure 10) illustrate cases where memory is read incorrectly, even if it may never have been written incorrectly. In scenario 9, one unit of work reads \(l_1\) followed by reading \(l_2\). Thus one thread is observing the state of multiple parts in the atomic set. If an update to the whole set is interleaved, then the values observed belong to different operations. The rest of the scenarios are problematic for a similar reason. These scenarios are similar to the “read skew” database anomaly.

All scenarios that only manipulate one memory location are marked as having a data race by the common definition. However, there are three scenarios missing \(R_u(l) R_{u'}(l) R_u(l), W_l(l) R_{u'}(l) R_u(l),\) and \(R_u(l) W_{u'}(l) W_l(l)\). None of these are problematic, but the common definition marks the last two as having a race. Our definition avoids these benign cases. An example of the third scenario is a thread performing \(x++\) and another printing the value of \(x\), where the write of \(x\) is atomic. This is non-deterministic but serializable, so there is no data race.

Not all database anomalies are applicable in this context. Some are concerned with an erroneous behavior when a transaction aborts and rolls back: e.g. “dirty read” and “dirty write”. Others refer to reading a set of memory locations that satisfy a search condition: “phantom read”. Finally the “write skew” anomaly is covered by several of our scenarios.

### 3.3 Completeness

We now show that the interleaving scenarios are complete, meaning that if an execution does not display them, then its projection on each atomic set is serializable, a concept that we define precisely below. To this end, we introduce a formal model of timestamps. Units of work can be totally ordered by the occurrence of their first write events in an execution. We associate a unique timestamp with each unit of work, respecting this order. A write event gets the timestamp of the unit of work to which it belongs. A read event gets the timestamp of the most recent write to the memory location it is reading. If a memory location gets written more than once by a unit of work, we mark the location as temporary, until the last write is completed. We use timestamps and temporary locations to capture consistency: if two reads within a unit of work get different timestamps, they are observing an inconsistent state. Likewise, observing a location marked as temporary by another unit of work is undesirable. We make the following assumption in our proof of completeness:

**Assumption 1.** We assume that each execution is such that every unit of work that writes some location in an atomic set, writes every location in that atomic set.

This assumption is not restrictive because we can always add “dummy writes” to any unit of work that does not satisfy it, and they are only needed conceptually.

In the rest of this section, we consider an execution \(E\) and its projection on some atomic set \(L\) in atomicSets(\(E\)). We call these the “execution” and the “projected execution”, respectively.

If the execution is not in accordance with the interleaving scenarios of Definition 1, then neither is the projection, and we show that the timestamp of writes to a given memory location in \(L\) are monotonically increasing (Lemma 1), and that no unit of work observes an inconsistent state (Lemma 2). These two properties suffice to show that the projected execution is serializable (Theorem 1), using the Serializability Theorem from database theory [8].

We use indices to refer to a total order of events in the projected execution\(^{3}\). The function \(event(i)\) gives the event at index \(i\). If \(u\) is a unit of work, then \(firstWrite(u)\) is the index of the first write event of \(u\).

We assume that timestamps are drawn from the natural numbers, and that the indices in an execution start at 1. We use \(ts(u)\) to denote the timestamp of a unit of work that performs writes. We allocate timestamps to units of work in such a way that:

\[
\text{ts}(u) < \text{ts}(u') \iff \text{firstWrite}(u) < \text{firstWrite}(u')
\]

So a unit of work \(u\), whose first write happens before the first write of another unit of work \(u'\) in an execution, gets a lower timestamp. Given a total order of timestamps thus allocated, let \(prev(t)\) be the timestamp immediately preceding \(t\) in this order (\(\text{prev}(t) < t\)).

We associate a timestamp, \(ts(i)\), with an event at index \(i\) in the execution. Write events get the timestamp of the unit of work to which they belong, and read events get the timestamp of the most recent write to the memory location read. \(ts(i)\) is computed as follows:

\[
\text{ts}(i) = \begin{cases} 
\text{ts}(u) & \text{if } event(i) = W_u(l) \\
\text{ts}(j) & \text{if } event(i) = R_u(l) \\
\& j < i \land event(j) = W_{u'}(l) \\
\& \exists k, j < k < i \land event(k) = W_{u''}(l) \\
0 & \text{if } event(i) = R_{u'}(l) \\
\& \exists j, j < i \land event(j) = W_{u''}(l)
\end{cases}
\]

So far \(ts(u)\) is only defined for units of work \(u\) that perform writes. For a unit of work that consists entirely of read events, let \(ts(u) = ts(i)\) for some \(i\) such that \(event(i) = R_{u'}(l)\). We will see in Lemma 2 that all such \(u\) have the same timestamp.

The predicate \(temp(u, i, l)\) is true if location \(l\) is temporary for unit of work \(u\) at index \(i\), meaning that there will be another write to \(l\) in \(u\) before \(i\). It is false for unit of work \(u\) at index \(i\) if \(i\) represents the index of the last write to \(l\) in \(u\). For reads, we take the value of \(temp(u, i, l)\) to be the value \(temp(u, i - 1, l)\). Initially,

---

\(^{3}\)A total order of events is natural for a sequentially consistent architecture. However, events happen in some total order even on weaker memory models, so our conceptual model is still applicable.
If the projected execution is not in accordance with any of the interleaving scenarios of Definition 1, its projection on each atomic set in atomicSets(\(E\)) is serializable.

**Theorem 1. Serializability** If the execution is not in accordance with any of the interleaving scenarios of Definition 1, its projection on each atomic set is serializable.

**Proof.** Assume that the execution is not in accordance with any of the interleaving scenarios of Definition 1, and that there exists an atomic set \(L\) in atomicSets(\(E\)), such that the projection of the execution on \(L\) is not serializable. We have that the projected execution is also not in accordance with the interleaving scenarios of Definition 1. By the Serializability Theorem [8], the conflict graph for the projected execution has a cycle: \(u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_n \rightarrow u_1\). Note that for any pair of events \(u, u'\) in this cycle, it cannot be the case that \(thread(u) = thread(u')\), because otherwise one of them would be nested inside the other, contradicting Observation 1. Consider two consecutive units of work \(u, u'\) in this cycle. Let \(i\) be the index of an event of \(u\) that conflicts with an event of \(u'\) with index \(j < i\). We show by cases that \(ts(u) \leq ts(u')\):

1. \(event(i) = W_u(l)\) and \(event(j) = W_u'(l)\). By Lemma 1, \(ts(i) < ts(j)\). Since \(ts(i) = ts(u)\) and \(ts(j) = ts(u')\), we have \(ts(u) \leq ts(u')\).
2. \(event(i) = W_u(l)\) and \(event(j) = R_u(l)\). We have \(ts(i) \leq ts(j)\). By Lemma 2, Part 2 and 3, \(ts(j)\) is either equal to \(ts(u')\) or \(prev(ts(u'))\). Also \(ts(i) = ts(u)\). Therefore we have either \(ts(u) \leq ts(u')\), or \(ts(u) \leq prev(ts(u'))\). Note that in the latter case \(ts(u) \leq ts(u')\), by the definition of prev.
3. \(event(i) = R_u(l)\) and \(event(j) = W_u'(l)\). We show by contradiction that \(ts(i) < ts(j)\). Assume first that \(ts(i) = ts(j)\). Then there must have been a \(k < i\) such that \(event(k) = W_u(l)\). So \(temp(u', i, l)\) is true. By Lemma 2, Part 1, we know that no event reads a location marked as temporary, so this is a contradiction and \(ts(i) \neq ts(j)\). Assume now that \(ts(i) > ts(j)\). In this case, there exists a \(k < i\), such that \(event(k) = W_{u'}(l)\) for some \(u'\), which is the write responsible for the read at \(i\). We have \(k < j\) and \(ts(k) > ts(j)\), which is a contradiction by Lemma 1. Therefore \(ts(i) < ts(j)\). Moreover, by Lemma 2, Part 2 and 3, \(ts(i)\) is either \(ts(u)\) or \(prev(ts(u))\), since \(ts(j) = ts(u)\), then we have either \(ts(u) < ts(u')\), or \(prev(ts(u)) < ts(u')\). Note that in the latter case \(ts(u) \leq ts(u')\), by the definition of prev.

So in all three cases, \(ts(u) \leq ts(u')\). So for our cycle \(u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_n \rightarrow u_1\), we have \(ts(u_1) \leq ts(u_2) \leq \cdots \leq ts(u_n) \leq ts(u_1)\). Therefore \(ts(u_1) = \cdots = ts(u_n)\). We know that the conflicting events of \(u_1\) through \(u_n\) must contain at least two writes from different units of work. This is the case because interleaving scenarios 2 and 3 do not happen in the execution, due of our initial assumption. Thus by the definition of \(ts(u)\), the fact that \(ts(u_1) = \cdots = ts(u_n)\) means that the execution is serializable.
in our approach, each class is responsible for its own synchronization and the projection of the execution on $L$

4. Implementation

This section presents an overview of the language constructs and their implementation for Java.

4.1 Overview of Language Constructs

In our approach, each class is responsible for its own synchronization by declaring one or more atomic sets. These declarations are inherited via subclassing, and subclasses may extend existing sets and/or introduce their own. The public and protected methods of a class are assumed to be units of work for its atomic sets, meaning that they preserve consistency when executed sequentially. If a unit of work accesses the elements of more than one atomic set in the same class, then it is guaranteed that no interleavings will occur in which other threads access any data in the union of these sets (though interleavings with other threads that only access unrelated data are allowed). The `unitfor` construct makes a client of a class to specify that a parameter needs to be manipulated atomically for the duration of its scope. If a method is already a unit of work for an atomic set $S$, then a `unitfor` declaration on parameter $p$ effectively makes the method a unit of work for the `union` of the set $S$ and the atomic sets of $p$.

The `owned(s)` construct is similar to `atomic(set)` but provides one level of indirection. It guarantees that the atomic set $a$ of the object pointed to by a field is included in the atomic set of the field. This mechanism allows transitively defined sets, and enables fine-grained concurrent access to recursive data structures. For example, the representation of a linked list may be included in an atomic set without also including the objects contained in the list. As part of future work, we will provide two variants of `owned` for arrays, one which includes the array itself, and the other which additionally includes the elements. Finally, a utility mechanism, the atomic class construct, helps making a class thread-safe, by declaring that all its fields are in a single atomic set. This avoids the need for synchronization wrappers in Java. Figure 12 summarizes the language constructs.

4.2 Synchronization Inference

We will now discuss an approach for generating code with synchronization that guarantees that the consistency properties declared using atomic set constructs are respected. In other words, that the problematic interleaving scenarios do not occur. First, we define a dataflow analysis over a program’s call graph that infers which locks need to be held for each unit of work. Then, we discuss how that information can be used to insert synchronization constructs.

Determining Atomic-Set Usage. The atomic sets that may be accessed by a unit of work can be determined by examining the code in the method that denotes the unit of work and in all methods transitively called by that method. In this set of methods, all field accesses are directly evident. The containment of fields in atomic sets is declared explicitly, so that computing the atomic sets accessed by each unit of work is straightforward. The analysis can be formulated as a standard dataflow problem using Kildall’s graph-based dataflow framework [26]. Recall that this framework associates sets $In(n)$ and $Out(n)$ with each node $n$ and defines the value $In(n)$ to be the union of values of all $Out(x)$ where the graph has an edge from $x$ to $n$. Node transfer functions define $Out(n)$ in terms of $In(n)$. We formulate a standard bit-vector problem, in which the bits are $read(s)$ and $write(s)$ for each atomic set $s$ in the program.

We define the dataflow problem across a standard dataflow graph $G = (N_G, E_G)$ that captures the dataflow among pointer values in the program. There is an $In$ and an $Out$ set for each value in $N_G$. The edges in our problem consist of: (i) the inverse of edges in $E_G$ and (ii) edges derived from reads of owned fields. The latter edges ensure that accesses to objects that are owned are treated as accesses of the owner set. The dataflow equations are shown in Figure 13, where the notation $[s]$ declares that the field $f$ is included in the atomic set $s$, and the notation $\{s\}$ denotes the fact that field $f$ is declared to be in atomic set $s$, and a statement $\{s\}$ means that a statement occurs in the method associated with node $n$. The result of the analysis is, for each pointer value $v$ in the program, a set $Out(v)$ of all atomic sets accessed from $v$ and from any $v'$ to which objects might transitively flow from $v$.

Adding Synchronization. For each method $m$, we acquire locks for all atomic sets that may access according to the above analysis and for which $m$ is a unit of work. This includes atomic sets that are accessed by methods $m'$ transitively called by $m$ as well as atomic sets accessed from fields transitively owned by elements in atomic sets accessed by $m$. Note that atomic sets accessed transitively from $m$ may include atomic sets declared in subclasses of the class that declares $m$.

Various kinds of locks can be used for synchronization. The most conservative strategy is to use exclusion, which prevents all problematic interleaving scenarios. We initially implemented this strategy using Java’s synchronized blocks. However, our problematic interleaving scenarios enables more aggressive implementations. In particular, we implemented the use of reader-writer locks [31] in which multiple readers are permitted concurrent access, but where writers must have exclusive access. Since all scenarios of Figure 9 involve at least one writer, this scheme is clearly correct. Potentially, we could analyze units of work for occurrences of problematic interleaving scenarios and generate customized synchronization that prohibits possible bad interactions.

Assumptions like having call graphs or global dataflow graphs make this implementation most suitable for whole-program compilation where relatively precise graphs can be constructed. How-

\[
\begin{align*}
In(v) & \leftarrow \{Out(v_i) \mid v \rightarrow v_i \in E_G \forall (3x_i = v.f \land \exists s \text{ owned}(s) f)\} \\
Out(v) & \leftarrow \{in(v) \cup \text{read}(v(s))\} \\
write(v) & \leftarrow \{write(s) \mid \exists v.f = x \in n \land f \in s\}
\end{align*}
\]

Figure 13. Dataflow equations for determining atomic sets accessed from pointers.
were 5 methods in Vectoring synchronization. For each such class, we manually removed all synchronized research.

However, it is possible to use approximations of unknown portions of the graph when the whole program is not available.

### 4.3 Experimental Results

We have implemented a prototype for synchronization inference using the Eclipse refactoring framework [6]. The inference engine is based on Domo [16], an analysis infrastructure developed at IBM Research.

Our language constructs are sufficient to correctly add synchronization to a significant subset of the Java Collections Framework. Figure 14 shows, for several classes in that framework, the number of original synchronization blocks, the number of atomic sets added, the number of owned fields and the methods requiring the unitfor construct. The notation n/a is used for classes that had no original synchronization because they were not intended for concurrent use.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>orig. sync.</th>
<th>added data-centric constructs</th>
<th>owned fields</th>
<th>unitfors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector</td>
<td>37</td>
<td>Vector(unitfor Collection)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>removeAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>retainAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HashMap</td>
<td>17</td>
<td>HashMap(unitfor Collection)</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>removeAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>retainAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Observable</td>
<td>8</td>
<td>Observable(unitfor Collection)</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>removeAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>retainAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random</td>
<td>3</td>
<td>Random(unitfor Collection)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>removeAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>retainAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ArrayList</td>
<td>n/a</td>
<td>ArrayList(unitfor Collection)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>removeAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>retainAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LinkedList</td>
<td>n/a</td>
<td>LinkedList(unitfor Collection)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>removeAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>retainAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HashSet</td>
<td>n/a</td>
<td>HashSet(unitfor Collection)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>removeAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>retainAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TreeSet</td>
<td>n/a</td>
<td>TreeSet(unitfor Collection)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>removeAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>retainAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HashMap</td>
<td>n/a</td>
<td>HashMap(unitfor Collection)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>removeAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>retainAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LinkedHashMap</td>
<td>n/a</td>
<td>LinkedHashMap(unitfor Collection)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>removeAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>retainAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TreeMap</td>
<td>n/a</td>
<td>TreeMap(unitfor Collection)</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addAll(unitfor Collection)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BitSet</td>
<td>n/a</td>
<td>BitSet(unitfor Collection)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>intersect(unitfor BitSet)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>equals(unitfor Object)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>and(unitfor BitSet)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>or(unitfor BitSet)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>xor(unitfor BitSet)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>andNot(unitfor BitSet)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 14. For each benchmark the table shows the number of original synchronization blocks, the number of atomic sets added, the number of owned fields and the methods requiring the unitfor construct. The notation n/a is used for classes that had no original synchronization because they were not intended for concurrent use.

5. Related Work

Most static [15, 28] and dynamic race detectors [30, 33], as well as type systems [9, 19] and languages [5] that guarantee race freedom are based on the common definition of data races and therefore do not handle high-level races. Type systems use redundant annotations to verify that data races do not occur. In contrast, our system infers the appropriate synchronization to prevent high-level as well as low-level data races, and does not require the programmer to keep track of locks explicitly.

An extension to ESC/Java detects a class of high-level data races, called “state-value errors” [10, 4]. The value of a local variable is stale if it is used beyond the critical section in which it was defined. Scenario 1 of our definition of data races addresses state-value errors. View consistency [3] is a correctness criterion that ensures that multiple reads in a thread observe a consistent state. A view is defined to be the set of variables that a lock protects. Two threads are view consistent if all the views in the execution of one, intersected with the maximal view of the other, form a chain under set inclusion. View consistency can be checked dynamically [3] or statically [34]. Scenarios 8 through 11 of our definition of data races address the issue of multiple reads getting an inconsistent state. In our approach, however, the programmer indicates explicitly what sets of locations form an atomic set, so this information does not need to be extracted from the locking structure of the code, which may not be correct.

Atomicity [21] is a non-interference property used to reason about multi-threaded programs. An atomic section can be assumed to execute serially without interleaved steps from other threads. A number of tools have been developed for checking atomicity violations, including type systems [21, 22, 18, 32]; dynamic analysis such as the Atomizer [20] which combines the theory of reduction [29] and ideas from dynamic race detectors; and model checking techniques [25, 17]. These approaches require atomicity annotations in addition to synchronized blocks from the programmer. In contrast, we have aimed at minimizing the amount of annotation required to specify synchronization constraints. Units of work are different from atomic code blocks in that they are related to the sequential, rather than concurrent, behavior of code and preserve the consistency of data when executed sequentially. They correspond naturally to method bodies in a well-designed object-oriented program. They are also data-centric because they are declared on specific atomic sets, which sometimes allows more concurrency than an atomic code block (see, e.g., method longDistanceTransfer() in Example 3).

Our definition of data races differs from the theory of reduction [20, 18], which provides a single pattern for atomicity, that is a sequence of right movers, followed by at most one atomic action. An atomic section can be assumed to execute serially without interleaved steps from other threads. All experiments took less than one minute on a 1.7GHz Pentium III with 768MB of memory. The first four lines refer to classes such as Vector which had existing synchronization. For each such class, we manually removed all synchronization blocks, and then added a single atomic set. There were 5 methods in Vector that needed the unitfor construct, and most of these correspond to published high-level data races that are easily avoided using our constructs. Observe that our approach generally requires far fewer annotations than the traditional approach.

For example, Vector requires only 1 atomic set, 1 owned field, and 5 unitfors instead of the original 37 synchronized blocks.

The rest of the benchmarks are classes that did not have synchronization. As the figure indicates, very few annotations are needed to make each of them thread-safe. This is to be contrasted with Java’s synchronization wrappers, such as e.g. SynchronizedSet, which wrap each method of the base class in a synchronization block. These wrappers are long and error-prone classes, since there is an explicit lock that must be held at the right places. With our constructs, synchronization wrappers are no longer needed.

This fragment of code is non-atomic, and can be fixed as follows:

```java
synchronized(lock) {
    synchronized(lock) { t = x; }
    t++;
}
```

Even though this fragment is now atomic, the theory of reduction would reject it, since it consists of a right mover, followed by two atomics, followed by a left mover. To overcome shortcomings of the underlying theory, the Atomizer tool [20] performs additional
analysis to determine reentrant locks, as well as protected locks. The type system of [18] remedies this problem by providing more precision via conditional atomicities. In contrast, our definition of data races is a complete set of non-serializable patterns, and is not based on locking structures. A tool based on our definition would not consider the above fixed code as problematic, because it would observe accesses to data rather than locks.

Language-level atomic sections [11] and software transactional memory [2, 24, 23, 36] are methods for removing the burden on the programmer in determining which locks to hold, by allowing code blocks to be marked as atomic. These code-centric approaches still require non-local reasoning from the programmer as illustrated in Section 2. A correct implementation of these methods needs to guarantee that there exists a global serial order of execution for the atomic sections. This is in general hard to implement efficiently in an imperative language, and requires specialized hardware [2]. The requirement for our units of work is that there exists a serial order only with respect to each atomic set, and there may not be a global serial order. By weakening the guarantee, while still maintaining correctness (preservation of data consistency), we have a method that is much easier to implement.

Our problematic interleaving scenarios are similar to those used by Wang and Stoller [35] to provide run-time analyses for atomicity. Our scenarios are simpler, and more importantly they are complete, meaning that an execution not displaying them is guaranteed to have a property related to serializability.

The scenarios in our definition of data races are analogous to anomalies used to characterize levels of isolation in databases, and defined in the ANSI SQL standard [37, 7]. Commercial databases allow programmers to trade off consistency for performance by offering different levels of isolation. Each level is characterized by the set of anomalies it does not allow. The highest level of isolation is serializability. Our problematic interleaving scenarios are similar to the schedules used to express the database anomalies. Some of these are not directly applicable in the context of concurrent programming, because they explicitly talk about a transaction committing or aborting.

Atomic sets share characteristics with data groups [27]. Data groups help in the specification of methods whose overrides may modify additional state introduced in subclasses. A method that is allowed to modify a data group is allowed to modify its downward closure, consisting of all member variables added in subclasses. Atomic sets are similar in that subclasses may add locations to a set declared in a parent class. They differ in that, unlike data groups, they are not hierarchical and non-overlapping.

The Serializability Violation Detector (SVD) [38] is a tool that dynamically infers atomic sections (called Computation Units or CUs), based on data and control dependences, and then detects if these CUs are non-serializable by checking a rule based on strict 2-Phase Locking. One of its key features is that it does not rely on the possibly buggy locking structure of the program to infer CUs. We share a similar viewpoint by having a definition of data races that does not rely on locks. SVD produces both false positives and false negatives, depending on the precision of the inferred CUs. It does not consider some of our interleaving scenarios to be problematic. This is always the case for Scenario 2, and some of the time for other scenarios because accesses can end up in different CUs when there is no data or control dependence between them.

Deng et al [14] present a method that allows the user to specify synchronization patterns that are used to synthesize synchronized code. The generated code can then be verified using the Bandera toolset. The user must specify explicitly the regions of code that need synchronization, but we do not require this. Unlike them, we only focus on one kind of synchronization pattern: exclusion between two regions that access the same atomic set.

The Actor model [1] defines objects that are updated atomically by individual methods. The Actor model shares our focus on using objects to manage consistency, but there are some crucial differences. First, it has a more restrictive notion of state changes, with a single become operation. Second, it is asynchronous, and does not have the notion of nesting of units of work. Third, this model does not support our notion of multiple consistency properties within a single object. Fourth, these languages lack a compositional structure like our own. Fifth, these languages do not support a construct such as unitfor for customizing consistency. Some Actor-based languages address some of these issues—Concurrent Aggregates [13] added synchronous calls and nesting, and ICC++ [12] had a limited form of composition with integral.

6. Conclusions

We presented a new definition of a data race as a collection of 11 problematic interleaving scenarios, which subsumes the traditional notion of a data race as well as high-level data races such as stale-value errors and inconsistent views. We have proved it complete by demonstrating that any execution that does not exhibit any of the 11 scenarios is equivalent to a serial execution, when projected onto each atomic set.

We have proposed a small number of language constructs that allow programmers to specify atomic sets, and a simple static analysis to determine the places in the code where synchronization is needed in order to avoid data races according to our new definition. Our data-centric approach is a declarative and succinct way for the programmer to specify synchronization constraints, in a way that maps naturally to the encapsulation provided by objects. It is less error-prone because the constructs are easy to use and the synchronization is inserted automatically.

The experiments indicate that these constructs suffice for much of the Java Collections Framework, and they also show greatly reduced annotations compared to synchronized blocks.

References


A. Proof of Lemma 1

Assume that the projected execution is not in accordance with any of the problematic interleaving scenarios of Definition 1. Assume that there exist \( i \) and \( j \) such that \( i < j \), \( \text{event}(i) = W_u(l) \), and \( \text{event}(j) = W_{u'}(l) \) for some \( l \) and \( u \neq u' \), but \( ts(i) \geq ts(j) \). Since \( u \neq u' \), then \( ts(i) \) and \( ts(j) \) could not be equal. So we have \( ts(i) > ts(j) \).

Since \( ts(i) = ts(u) \), and \( ts(j) = ts(u') \), it must be that the first write of \( u \) occurs after the first write of \( u' \). Let \( i' = \text{firstWrite}(u) \) and \( j' = \text{firstWrite}(u') \). Then we have that \( j' < i' < j \).

Case 1. \( \text{event}(j') = W_{u'}(l) \). It must be that \( \text{thread}(u) \neq \text{thread}(u') \), because otherwise \( u \) and \( u' \) would be nested units of work and this would contradict Observation 1 (two non-nested units of work of the same thread do not have interleaved events). Then the projected execution is in accordance with the interleaving scenario 5 of Definition 1, which is a contradiction.

Case 2. \( \text{event}(j') = W_u(l') \), \( l' \neq l \). Since unit of work \( u \) also writes \( l' \), by Assumption 1, the index \( k \) of such a write is greater than \( i' \), since \( i' \) is the index of the first write of \( u \). We have that \( \text{thread}(u) \neq \text{thread}(u') \) for the same reason as above. Therefore the projected execution is in accordance with one of the interleaving scenarios 6 or 7, which is a contradiction.

B. Proof of Lemma 2

Table 1 illustrates the different cases appearing in this proof.

Part 1. Assume that the projected execution is not in accordance with the interleaving scenarios of Definition 1. Consider an index \( i \) such that \( \text{event}(i) = R_u(l) \), and \( \text{temp}(u', i, l) = \text{true} \) for some \( u' \neq u \) such that \( \text{thread}(u) \neq \text{thread}(u') \). Then there must be a \( j \) and \( k \), \( j < i < k \), such that \( \text{event}(j) = \text{event}(k) = W_{u'}(l) \). But the projected execution would be in accordance with interleaving scenario 3 (Table 1), which is a contradiction. So \( \text{temp}(u', i, l) = \text{false} \).

Part 2. Assume that the projected execution is not in accordance with the interleaving scenarios of Definition 1. Consider a unit of work \( u \) that contains at least a read and a write event. Assume that there is an \( i \), such that \( \text{event}(i) = R_u(l) \), and \( ts(i) \notin \{ts(u), \text{prev}(ts(u))\} \). Let \( k, k < i \), be the index of the write responsible for the value of \( ts(i) \). So \( \text{event}(k) = W_{u'}(l) \) for some \( u' \).

Case 1. \( ts(i) > ts(u) \). Since unit of work \( u \) must also write \( l \) by Assumption 1, let \( j \) the index of this write, \( \text{event}(j) = W_u(l) \). Note that we cannot have \( k < j < i \), since the write at index \( k \) is responsible for the value of \( ts(i) \).

Subcase 1a. \( j < k < i \)

We have that \( \text{thread}(u) \neq \text{thread}(u') \), because otherwise \( u \) and \( u' \) would have to be nested units of work, and this would contradict Observation 1. Then the projected execution is in accordance with interleaving scenario 4 (Table 1), which is a contradiction.
Subcase 1b. \( k < i < j \)

We have that \( ts(i) = ts(k) > ts(u) = ts(j) \), which contradicts Lemma 1.

Case 2. \( ts(i) < \text{prev}(ts(u)) \). There must be a \( j \) such that \( \text{event}(j) = W_{\alpha}(l) \) by Assumption 1, and \( ts(j) = \text{prev}(ts(u)) \). Since \( ts(k) = ts(i) < \text{prev}(ts(u)) \), then it cannot be the case that \( j < k \), because otherwise that would contradict Lemma 1. We also have that \( k > i \), otherwise that would contradict Lemma 1. Therefore we have:

\[
\begin{align*}
W_{\alpha}(l) &< i < j \\
W_{\alpha}(l) &< R_u(l) < W_{\alpha}(l)
\end{align*}
\]

There must be an index \( i' \) such that \( \text{event}(i') = W_{\alpha}(l) \) by Assumption 1. We have that \( i' > j \), because otherwise that would contradict Lemma 1. We have that \( thread(u) \neq thread(u') \), because otherwise that would contradict Observation 1. Thus the execution is in accordance with the interleaving scenario: \( \text{prev}(ts(u)) \) \( R_u(l) \) \( W_{\alpha}(l) \) \( W_{\alpha}(l) \) (Table 1), which is scenario 1 from Definition 1. This is a contradiction. Therefore \( ts(i) \in \{ ts(u), \text{prev}(ts(u)) \} \).

Part 3. Assume that the projected execution is not in accordance with any of the interleaving scenarios of Definition 1. Assume that there exists a unit of work \( u \), and \( i < j \), such that \( \text{event}(i) = R_u(l) \), \( \text{event}(j) = R_u(l') \), and \( ts(i) \neq ts(j) \). Suppose that the unit of work \( u \) does not contain any writes. Let \( i' \) and \( j' \) be the indices of writes responsible for the values of \( ts(i) \) and \( ts(j) \). We have that \( i' < i \) and \( j' < j \), and \( \text{event}(i') = W_{\alpha}(l), \text{event}(j') = W_{\alpha}(l') \).

Case 1. \( l = l' \). Then it must be that:

\[
\begin{align*}
W_{\alpha}(l) &< i < j \\
R_u(l) &< W_{\alpha}(l) < R_u(l')
\end{align*}
\]

We have \( thread(u) \neq thread(u') \) because otherwise that would contradict Observation 1. But then the execution is in accordance with interleaving scenario 2 of Definition 1 (Table 1), which is a contradiction.

Case 2. \( l \neq l' \) and \( ts(i) < ts(j) \).

Case 2a. \( \text{event}(i') = W_{\alpha}(l), \text{event}(j') = W_{\alpha}(l') \)

The unit of work \( u'' \) must write \( l \) as well by Assumption 1. Let \( k \) be the index of such a write, \( \text{event}(k) = W_{\alpha}(l) \). It cannot be the case that \( k < i' \), because that would contradict Lemma 1. So \( k > i' \), since the write at \( i' \) is the one responsible for the value at \( i \). We have that \( thread(u) \neq thread(u') \), because otherwise that would contradict Observation 1. Thus the execution is in accordance with one the interleaving scenarios 9 or 10 (Table 1), which is a contradiction.

Case 2b. \( \text{event}(k) = W_{\alpha}(l) \). Since \( ts(i') = ts(i) < ts(j) = ts(j') = ts(k) \), then it must be that \( k > i' \), because otherwise that would contradict Lemma 1. We also have that \( k > j' \) because \( i' \) is the index responsible for the value read at \( i \). We have that \( thread(u) \neq thread(u') \), because otherwise that would contradict Observation 1. Therefore the execution is in accordance with one of interleaving scenarios 8 and 11 (Table 1), which is a contradiction.

Case 3. \( l \neq l' \) and \( ts(i) > ts(j) \). The unit of work \( u' \) must write \( l' \) as well by Assumption 1. Let \( k \) be the index of such a write, \( \text{event}(k) = W_{\alpha}(l') \). Since \( ts(u') = ts(i) > ts(j) = ts(u'') \), then it must be that \( k > j' \), because otherwise that would contradict Lemma 1. Since \( j' \) is the index of the write responsible for the read at \( j \), then it is also the case that \( k > j \). We have that \( thread(u) \neq thread(u') \), because otherwise that would contradict Observation 1. Therefore the execution is in accordance with the interleaving scenario 8 (Table 1), which is a contradiction.

Therefore \( ts(i) = ts(j) \).