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.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming-parallel programming; D.2.4 [*Software Engineering*]: Software/Program Verification-reliability; F.1.3 [*Logics And Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Theory

Keywords Concurrent Object-Oriented Programming, Data Races, Serializability, Programming Model

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

* This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL '06 January 11–13, 2006, Charleston, South Carolina, USA.

Copyright © 2006 ACM 1-59593-027-2/06/0001... $5.00.

```
class Customer {
  String city;
  int zipcode;
  Date date;
  Item item;

  void updateAddress(String c, int z){
    atomic { city = c; zipcode = z; }
  }
  void newPurchase(Date d, Item i){
    atomic { date = d; item = i; }
  }
}
class PreferredCustomer extends Customer {
  void newStoreGift(Date d, Item i){
    atomic { date = d; item = i; }
  }
}
```
(a)

```
class Customer {
  atomic(address) String city;
  atomic(address) int zipcode;
  atomic(purchase) Date date;
  atomic(purchase) Item item;

  void updateAddress(String c, int z){
    city = c; zipcode = z;
  }
  void newPurchase(Date d, Item i){
    date = d; item = i;
  }
}
class PreferredCustomer extends Customer {
  void newStoreGift(Date d, Item i){
    date = d; item = i;
  }
}
```
(b)

Figure 1. Customer example.

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 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[1]. 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 $\mathit{Units}(A)$. 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 $u \in \mathit{Units}(A)$, the guarantee is that any pair

[1] 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 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 either of these methods to be interleaved with that of method 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 declares those fields. However, there are situations where additional synchronization on parameters is needed.

Figure 2(a) shows a fragment of class java.util.Vector from the Java Standard Collections Framework. Specifically, the figure shows the declaration of a field elementData, which refers to the array that stores the vector's contents, and a field 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 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 k. The race occurs if a thread that executes the constructor's code is interrupted after executing the statement elementCount = c.size() by another thread that is calling the removeAllElements() method on the collection pointed to by c. Then, when the first thread resumes, and executes the statement c.toArray(elementData), the resulting vector will contain 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 elementCount and elementData have been placed in an atomic set vec, and the constructor has been designated as a unit of work for its parameter c. Note that only the field elementData is in the atomic set and not the vector. The unitfor construct used in this example is a mechanism for specifying client-side synchronization constraints, and declares that the scope of parameter c is a unit of work for all atomic sets of 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 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 Account and Bank. Account has a field checking and methods withdraw() and 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. Bank provides a method transfer() for transferring money between accounts, and declares fields log and logCount for maintaining a log of completed transfers. Observe that log and logCount have been placed in an atomic set 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 assume that a distinction needs to be made between local transfers, 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 distinction has been encoded by two methods, localTransfer() and longDistanceTransfer(), both of which invoke the previously discussed transfer() method. In essence, we would like to express that localTransfer() is a unit of work for its parameters a and b, and this is accomplished using the unitfor construct. As localTransfer() reads both a and b, synchronization will be inserted to ensure that the call to transfer() will be executed atomically.

Observe that this solution allows for more concurrency than a traditional solution where the body of the 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 a'.deposit(m) and log'.add(c,d,m) can be interleaved with calls to localTransfer() and LongDistanceTransfer(), respectively. Note that calls to deposit() can be interleaved with calls to longDistanceTransfer() while preserving the consistency of log and logCount.

```
class Account {
  atomic(account) int checking;
  public void deposit(int n) { ··· }
  public void withdraw(int n) { ··· }
}
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.

2.4 Example 4: Synchronization Wrappers

The Java Collections Framework provides *synchronization wrappers* for creating synchronized versions of collections that are not thread-safe. For example, class java.util.ArrayList provides array-based lists that are not thread-safe. An application that wishes to use a thread-safe ArrayList typically executes code such as:

```
List myList =
  Collections.synchronizedList(new ArrayList())
```

Here, the synchronizedList() method from the utility class java.util.Collections creates a decorator object of type List that wraps the ArrayList that was passed in as a parameter, and that forwards all methods to this ArrayList. All forwarding methods are synchronized, thus preventing low-level data races that might otherwise be caused by concurrent accesses to methods such as get() and set(). Note that this only prevents races when

```
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);
  }
}
```

(a)

```
public class Vector {
  atomic(vec) Object[] elementData;
  atomic(vec) int elementCount;

  public Vector(unitfor 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);
  }
}
```

(b)

Figure 2. Vector example.

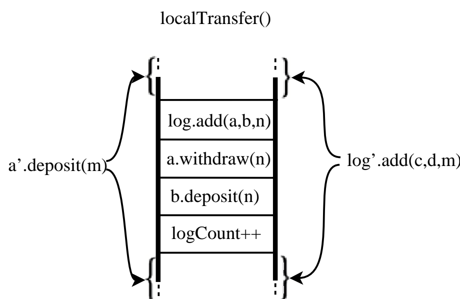

Figure 4. Allowable interleavings for `localTransfer`. Arrows indicate where `a'.deposit(m)` and `log'.add(c,d,n)` can be interleaved, assuming that `a` and `a'` and `log` and `log'` may point to the same objects, respectively.

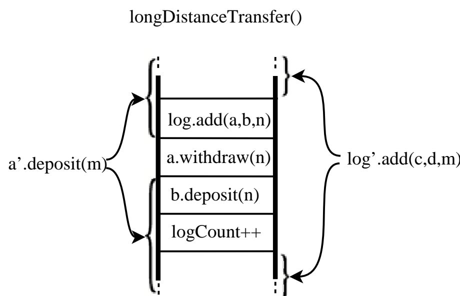

Figure 5. Allowable interleavings for `longDistanceTransfer`. Arrows indicate where `a'.deposit(m)` and `log'.add(c,d,n)` can be interleaved, assuming that `a` and `a'` and `log` and `log'` may point to the same objects, respectively.

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—*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 `atomic` at allocation sites. For example, a thread-safe `ArrayList` can be created as follows:

```
List myList = new atomic ArrayList(){};
```

This eliminates the need for the synchronization wrapper classes that contain large numbers of boilerplate forwarding methods, except for those few that need `unitfor` parameters.

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

Figure 6. Linked List Example

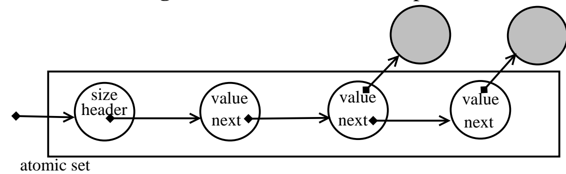

Figure 7. A linked list. Only the objects in the representation of linked list are contained in an atomic set, not the objects contained in the list (shaded).

2.5 Example 5: Owned Fields

The `atomic` set construct is used to include fields in an atomic set. Sometimes it is useful to reason about atomic sets of objects referred to by a field. In the example of Figure 6, a linked list class has two fields `header` and `size` that belong to atomic set `list`. Field `header` is of type `Entry`, which declares its own atomic set `entry`. Method `set()` takes an index in the list, finds the proper 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.

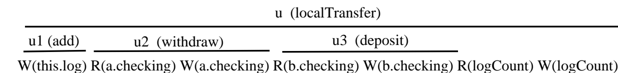

Figure 8. Units of work and accesses in `localTransfer`

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 $\mathcal{L}$ be the set of all memory locations. A subset $L \subseteq \mathcal{L}$ may be designated as *atomic*. An *event* is an access to a memory location $l \in \mathcal{L}$. 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 $L - l$ 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 $\mathit{sets}(u)$ for the set of atomic sets corresponding to u. We say that $\bigcup_{L \in \mathit{sets}(u)} L$ 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 $L \in \mathit{sets}(u)$. 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 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 $\mathit{thread}(u)$ 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.

	Interleaving scenario	Description
1.	$R_u(l)\ W_{u'}(l)\ W_u(l)$	Value read is stale by the time an update is made in u.
2.	$R_u(l)\ W_{u'}(l)\ R_u(l)$	Two reads of the same location yield different values in u.
3.	$W_u(l)\ R_{u'}(l)\ W_u(l)$	An intermediate state is observed by u'.
4.	$W_u(l)\ W_{u'}(l)\ R_u(l)$	Value read is not the same as the one written last in u.
5.	$W_u(l)\ W_{u'}(l)\ W_u(l)$	Value written by u' is lost.
6.	$W_u(l_1)\ W_{u'}(l)\ W_{u'}(L{-}l)\ W_u(l_2)$	Memory is left in an inconsistent state.
7.	$W_u(l_1)\ W_{u'}(l_2)\ W_u(l_2)\ W_{u'}(l_1)$	same as above.
8.	$W_u(l_1)\ R_{u'}(l)\ R_{u'}(L{-}l)\ W_u(l_2)$	State observed is inconsistent.
9.	$R_u(l_1)\ W_{u'}(l)\ W_{u'}(L{-}1)\ R_u(l_2)$	same as above.
10.	$R_u(l_1)\ W_{u'}(l_2)\ R_u(l_2)\ W_{u'}(l_1)$	same as above.
11.	$W_u(l_1)\ R_{u'}(l_2)\ W_u(l_2)\ R_{u'}(l_1)$	same as above.

Figure 9. Problematic Interleaving Scenarios. These scenarios are complete provided that each unit of work that writes to an atomic set, writes all locations in that set.

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 $\mathit{thread}(u) = \mathit{thread}(u')$ 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, $R_u(l)\ W_{u'}(l)\ W_u(l)$ 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 u.

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, $\mathit{atomicSets}(E)$, 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 $\mathit{thread}(u) \neq \mathit{thread}(u')$. 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 `x++`. Scenario 1 corresponds roughly to the "lost update" [37] anomaly in databases:

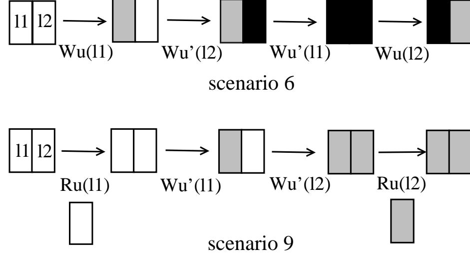

Figure 10. Problematic interleaving scenarios

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 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 inconsistently, 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 $R_u(l)\ R_{u'}(l)\ R_u(l)$, $W_u(l)\ R_{u'}(l)\ R_u(l)$, and $R_u(l)\ R_{u'}(l)\ W_u(l)$. 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[2]. 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:

$$\mathit{ts}(u) < \mathit{ts}(u') \Leftrightarrow \mathit{firstWrite}(u) < \mathit{firstWrite}(u').$$

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 (*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:

$$\mathit{ts}(i) = \begin{cases} \mathit{ts}(u) & \text{if } \mathit{event}(i) = W_u(l) \\ \mathit{ts}(j) & \text{if } \mathit{event}(i) = R_u(l) \\ & \wedge\ j < i \wedge \mathit{event}(j) = W_{u'}(l) \\ & \wedge\ \nexists k, j < k < i \mid \mathit{event}(k) = W_{u''}(l) \\ 0 & \text{if } \mathit{event}(i) = R_u(l) \\ & \wedge\ \nexists j, j < i \mid \mathit{event}(j) = W_{u'}(l) \end{cases}$$

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 i have the same timestamp.

[2] 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.

index i	event	$ts(i)$	$temp(u_1, i, l_1)$	$temp(u_1, i, l_2)$ $temp(u_2, i, l_1)$ $temp(u_2, i, l_2)$
1	$W_{u_1}(l_1)$	1	*true*	*false*
2	$R_{u_2}(l_1)$	1	*true*	*false*
3	$R_{u_2}(l_2)$	0	*true*	*false*
4	$W_{u_1}(l_2)$	1	*true*	*false*
5	$W_{u_2}(l_1)$	2	*true*	*false*
6	$W_{u_2}(l_2)$	2	*true*	*false*
7	$W_{u_1}(l_1)$	1	*false*	*false*

Figure 11. Sample execution and timestamps

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 beyond index 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, $temp(u, 0, l) = false$ for all u and l. It is computed as follows:

$$temp(u,i,l) = \begin{cases} true & \text{if } event(i) = W_u(l) \land \\ & \exists\, j > i \mid event(j) = W_u(l) \\ false & \text{if } event(i) = W_u(l) \land \\ & \nexists\, j > i \mid event(j) = W_u(l) \\ temp(u, i-1, l) & \text{otherwise.} \end{cases}$$

Figure 11 gives a sample execution and its timestamps. In this example, there are two units of work u_1 and u_2 in different threads, and two locations l_1 and l_2. We have $ts(u_1) = 1$ and $ts(u_2) = 2$. At index 2, unit of work u_2 reads an intermediate value of location l_1. This is captured by $temp(u_1, 2, l_1)$ being true.

The following lemma states that the timestamps of write events on the same memory location l are monotonically increasing.

Lemma 1. If the projected execution is not in accordance with the interleaving scenarios of Definition 1, and i and j are such that $i < j$, $event(i) = W_u(l)$ and $event(j) = W_{u'}(l)$ for some l, $u \neq u'$, then $ts(i) < ts(j)$.

The proof of Lemma 1 can be found in Appendix A.

The following lemma states that the state observed in a unit of work is consistent, by giving three properties of read events in an execution that is not in accordance with any of the scenarios in Definition 1. First, no temporary value is ever read. Second, reads in a unit of work that also writes the same atomic set do not get stale values. Third, two reads in a unit of work that does not perform any writes to same atomic set get consistent values.

Lemma 2. If the projected execution is not in accordance with any of the interleaving scenarios of Definition 1:

1. No event from one unit of work reads a memory location marked as temporary by another unit of work in a different thread.
   $\forall u, i, l \mid event(i) = R_u(l) \Rightarrow \nexists u' \mid u' \neq u \land thread(u) \neq thread\,(u') \land temp(u', i, l)$.
2. A read in a unit of work that also contains a write to the same atomic set does not get a stale value, i.e. it gets the timestamp corresponding to the unit of work or the previous one.
   $\forall\, i \mid (event(i) = R_u(l) \land \exists\, j \mid event(j) = W_u(l)) \Rightarrow ts(i) \in \{ts(u), prev(ts(u))\}$.
3. Reads in a unit of work u that does not contain writes, get the same timestamp.
   $\forall\, i, j \mid (event(i) = R_u(l) \land event(j) = R_u(l') \land \nexists\, k \mid event(k) = W_u(l)) \Rightarrow ts(i) = ts(j) = ts(u)$.

The proof for Lemma 2 can be found in Appendix B.

Finally, we show that for an execution that is not in accordance with any of the interleaving scenarios of Definition 1, its projection on each atomic set is serializable, which we define precisely below. We can think of a unit of work as being a single-threaded transaction that always commits, and this allows us to use concepts from serializability theory [8]. Given indices i and i' such that $i < i'$, the pair ($event(i)$,$event(i')$) is a *conflicting pair of events*, if they are on the same memory location, and one of them is a write. We say that two executions are *equivalent* if they consist of the same units of work and the same events, and have the same pairs of conflicting events. An execution is *serial* if for every two units of work u and u' that appear in it, either all events in u happen before all events in u', or vice versa. We say that an execution is *serializable* if it is equivalent to an execution that is serial.

The *conflict graph* of an execution is a directed graph, with nodes consisting of units of work. There is an edge between units of work u and u', if u and u' have events e and e', respectively, such that (e,e') is a conflicting pair. The Serializability Theorem [8] states that an execution is serializable, if and only if its conflict graph is acyclic. We will use this fact to prove Theorem 1 below.

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 in $atomicSets(E)$ 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*, 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 u and 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 and u' in this cycle. Let i be the index of an event of u that conflicts with an event of u' with index j ($i < j$). 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 $ts(u) < ts(u')$.
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

Construct	Usage
`atomic(s)`	fields that have a consistency property or fields whose intermediate states should not be visible
`unitfor`	parameter that must be manipulated atomically
`atomic`	class that needs to be thread-safe
`owned(s)`	similar to `atomic(s)` but with one level of indirection through field dereference

Figure 12. Summary of language constructs

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 $ts(u_1) = \cdots = ts(u_n)$ is a contradiction. Therefore such a cycle does not exist and the projection of the execution on L is serializable. □

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. An atomic set declaration in a class means that each instance of that class has its own separate atomic set. 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. We assume that each access to a location in an atomic set is done within a unit of work for that atomic set[3]. 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 enables 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 `s` 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

[3] This assumption means that our system expects client-side field accesses to be done via getter/setter methods.

$$
\begin{aligned}
\mathit{In}(v) &\leftarrow \{\mathit{Out}(v_i) \mid\ v \rightarrow v_i \in E_G \vee \\
&\qquad (\exists \mathtt{v_i} = \mathtt{v.f} \wedge \exists \mathtt{s}\ \mathtt{owned(s)}\ f)\} \\
\mathit{Out}(v) &\leftarrow \mathit{In}(v) \cup \mathit{reads}(\mathrm{v}) \cup \mathit{writes}(\mathrm{v}) \\
\mathit{reads}(\mathrm{v}) &\leftarrow \{read(s) \mid \exists \mathtt{x} = \mathtt{v.f} \in n \wedge f \in s\} \\
\mathit{writes}(\mathrm{v}) &\leftarrow \{write(s) \mid \exists \mathtt{v.f} = \mathtt{x} \in n \wedge f \in s\}
\end{aligned}
$$

Figure 13. Dataflow equations for determining atomic sets accessed from pointers.

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[4]. 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 $\mathit{In}(n)$ and $\mathit{Out}(n)$ with each node n and defines the value $\mathit{In}(n)$ to be the union of values of all $\mathit{Out}(x)$ where the graph has an edge from x to n. Node transfer functions define $\mathit{Out}(n)$ in terms of $\mathit{In}(n)$. 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 $f \in s$ is used to denote the fact that field f is declared to be in atomic set s, and `statement` $\in n$ to mean 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 $\mathit{Out}(v)$ of all atomic sets accessed from v and from any v' to which objects might transitively flow from v.

Adding Synchronization. We associate a lock with each atomic set. For each method m, we acquire locks for all atomic sets that m may access according to the above analysis and for which m is a unit of work[5]. 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 enable 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

[4] We ignore the issue of fields accessed via mechanisms such as Java reflection. In such cases, we would need to use a conservative approximation of what fields might be accessed.

[5] We assume some ability to atomically acquire multiple locks, which is straightforward for locks that support POSIX-style trylock.

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

Deadlock When attempting to acquire locks for all the atomic sets that a method accesses, our approach consists of trying to acquire them all, and release them all if at least one is unavailable, and then trying again. This mitigates deadlock to some extent. Deadlock may still occur in the generated code if there are (transitive) cyclical dependences between the sets of locks needed by two units of work. This can be detected through static analysis. Future work includes building such an analysis to warn the programmer.

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 the Java Collections Framework[6]. Figure 14 shows, for several classes in that framework, the number of original `synchronized` blocks, the number of atomic sets needed, the number of owned fields, and the number of methods for which `unitfor` was needed. 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 `unitfor` constructs 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.

	orig.	added data-centric constructs		
Benchmark	**sync.**	**sets**	**owned**	**unitfors**
Vector	37	1	1	Vector(unitfor Collection) addAll(unitfor Collection) addAll(int,unitfor Collection) removeAll(unitfor Collection) retainAll(unitfor Collection)
Hashtable	17	1	3	void putAll(unitfor Map)
Observable	8	1	1	
Random	3	1	0	
ArrayList	n/a	1	1	ArrayList(unitfor Collection) addAll(unitfor Collection) addAll(int,unitfor Collection)
LinkedList	n/a	1	3	addAll(int,unitfor Collection)
SubList	n/a	1	0	addAll(int,unitfor Collection)
HashSet	n/a	1	0	HashSet(unitfor Collection)
TreeSet	n/a	1	1	addAll(unitfor Collection)
HashMap	n/a	1	3	
LinkedHashMap	n/a	1	3	
IdentityHashMap	n/a	1	1	putAll(unitfor Map) equals(unitfor Object)
TreeMap	n/a	1	4	putAll(unitfor Map)
BitSet	n/a	1	0	intersects(unitfor BitSet) equals(unitfor Object) and (unitfor BitSet) or (unitfor BitSet) xor (unitfor BitSet) andNot (unitfor BitSet)

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 "stale-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 stale-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

[6] Some limitations in our current implementation (most notably in handling inner classes) prevent us for performing the experiment on the entire Collections Framework.

sequence of right movers, followed by at most one atomic action, followed by a sequence of left movers. Lock acquires (releases) are considered right (left) movers. Consider the following fragment of code, where x is a shared variable and t is local:

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

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

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

Acknowledgments

We thank David Bacon, Rastislav Bodik, Stephen Fink, Robert O'Callahan, and Vivek Sarkar for very useful discussions.

References

[1] Gul Agha. An overview of actor languages. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, pages 58–67, New York, NY, USA, 1986.

[2] C. S. Ananian and M. Rinard. Language-level transactions. In *High-Performance Embedded Computing (HPEC)*, 2004.

[3] C. Artho, K. Havelund, and A. Biere. High-level data races. In *Proc. NDDL/VVEIS'03*, pages 82–93, 2003.

[4] C. Artho, K. Havelund, and A. Biere. Using block-local atomicity to detect stale-value concurrency errors. In *ATVA'04*, pages 150–164, 2004.

[5] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *Proc. OOPSLA'00*, pages 382–400, 2000.

[6] D. Bäumer, E. Gamma, and Adam Kieżun. Integrating refactoring support into a Java development tool. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) Companion*, October 2001.

[7] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. ACM SIGMOD Conf.*, pages 1–10, 1995.

[8] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[9] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proc. OOPSLA'01*, October 2001.

[10] M. Burrows and K. R. M. Leino. Finding stale-value errors in concurrent programs. Technical Report 2002-004, SRC, May 2002.

[11] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Vijay Saraswat, Vivek Sarkar, and Christoph von Praun. X10: An object-oriented approach to non-uniform cluster computing. In *Proc. OOPSLA'05*, San Diego, CA, 2005. To appear.

[12] A. Chien, U. Reddy, J. Plevyak, and J. Dolby. ICC++ — A C++ dialect for high performance parallel computing. *Lecture Notes in Computer Science*, 1049:76–95, 1996.

[13] Andrew A. Chien and William J. Dally. Concurrent aggregates (ca). In *Proc. PPoPP '90*, pages 187–196, 1990.

[14] X. Deng, M. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proc. ICSE'02*, May 2002.

[15] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proc. SOSP'03*, pages 237–252, October 2003.

[16] S. Fink, J. Dolby, , and L. Colby. Semi-automatic J2EE transaction configuration. Technical Report RC23326, IBM T.J. Watson Research Center, March 2004.

[17] C. Flanagan. Verifying commit-atomicity using model checking. In *Proc. SPIN'04*, pages 252–266, 2004.

[18] C. Flanagan, S. Freund, and M. Lifshin. Type inference for atomicity. In *Proc. TLDI'05*, pages 47–58, 2005.

[19] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proc. PLDI'00*, pages 219–232, 2000.

[20] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. POPL'04*, pages 256–267, 2004.

[21] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proc. PLDI'03*, pages 338–349, 2003.

[22] C. Flanagan and S. Qadeer. Types for atomicity. In *Proc. TLDI'03*, pages 1–12, 2003.

[23] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. OOPSLA'03*, pages 388–402, 2003.

[24] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proc. PPoPP'05*, 2005.

[25] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model checking. In *Proc. VMCAI'04*, pages 175–190, 2004.

[26] Gary A. Kildall. A unified approach to global program optimization. In *Proc. POPL'73*, pages 194–206, 1973.

[27] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *Proc. OOPSLA'98*, pages 144–153, 1998.

[28] K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. Technical Report 002, Compaq SRC, 1999.

[29] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *CACM*, 18(12), 1975.

[30] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proc. PPoPP'03*, pages 167–178, 2003.

[31] Java Community Process. JSR 166: Concurrency utilities. See `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`., September 2004.

[32] A. Sasturkar, R. Agarwal, L. Wang, and S. Stoller. Automated type-based analysis of data races and atomicity. In *Proc. PPoPP'05*, 2005.

[33] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *Proc. SOSP'97*, pages 27–37, October 1997.

[34] C. von Praun and T. Gross. Static detection of atomicity violations in object-oriented programs. *Journal of Object Technology*, 3(2), 2004.

[35] Liqiang Wang and Scott D. Stoller. Runtime analysis for atomicity for multi-threaded programs. Technical Report DAR-04-14, State University of New York At Stony Brook, May 2005.

[36] A. Welc, S. Jagannathan, and A. Hosking. Transactional monitors for concurrent objects. In *Proc. ECOOP'04*, pages 519–542, 2004.

[37] ANSI X3.135-1992. In *American National Standard for Information Systems – Database Language – SQL*, November 1992.

[38] M. Xu, R. Bodik, and M. Hill. A serializability violation detector for shared-memory server programs. In *Proc. PLDI'05*, pages 1–14, 2005.

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 $i < j$, $\mathit{event}(i) = W_u(l)$, and $\mathit{event}(j) = W_{u'}(l)$ for some l and $u \neq u'$, but $\mathit{ts}(i) \geq \mathit{ts}(j)$. Since $u \neq u'$, then $\mathit{ts}(i)$ and $\mathit{ts}(j)$ could not be equal. So we have $\mathit{ts}(i) > \mathit{ts}(j)$.

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

Case 1. $\mathit{event}(j') = W_{u'}(l)$. It must be that $\mathit{thread}(u) \neq \mathit{thread}(u')$, 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.

$$\overset{j'}{W_{u'}(l)} \quad \overset{i}{W_u(l)} \quad \overset{j}{W_{u'}(l)} \quad \text{(scenario 5)}$$

Case 2. $\mathit{event}(j') = W_{u'}(l')$, $l' \neq l$. 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 $\mathit{thread}(u) \neq \mathit{thread}(u')$ 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.

$$\overset{j'}{W_{u'}(l')} \quad \overset{k}{W_u(l')} \quad \overset{i}{W_u(l)} \quad \overset{j}{W_{u'}(l)} \quad \text{(scenario 6)}$$

$$\overset{j'}{W_{u'}(l')} \quad \overset{i}{W_u(l)} \quad \overset{k}{W_u(l')} \quad \overset{j}{W_{u'}(l)} \quad \text{(scenario 6)}$$

$$\overset{j'}{W_{u'}(l')} \quad \overset{i}{W_u(l)} \quad \overset{j}{W_{u'}(l)} \quad \overset{k}{W_u(l')} \quad \text{(scenario 7)}$$

Therefore $\mathit{ts}(i) < \mathit{ts}(j)$ as required.

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 $\mathit{event}(i) = R_u(l)$, and $\mathit{temp}(u', i, l) = \mathit{true}$ for some $u' \neq u$ such that $\mathit{thread}(u) \neq \mathit{thread}(u')$. Then there must be a j and k, $j < i < k$, such that $\mathit{event}(j) = \mathit{event}(k) = W_{u'}(l)$. But the projected execution would be in accordance with interleaving

scenario 3 (Table 1), which is a contradiction. So $\mathit{temp}(u', i, l) = \mathit{false}$.

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 $\mathit{event}(i) = R_u(l)$, and $\mathit{ts}(i) \notin \{\mathit{ts}(u), \mathit{prev}(\mathit{ts}(u))\}$. Let k, $k < i$, be the index of the write responsible for the value of $\mathit{ts}(i)$. So $\mathit{event}(k) = W_{u'}(l)$ for some u'.

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

Subcase 1a. $\overset{j}{W_u(l)} < \overset{k}{W_{u'}(l)} < \overset{i}{R_u(l)}$

We have that $\mathit{thread}(u) \neq \mathit{thread}(u')$, 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. $\overset{k}{W_{u'}(l)} < \overset{i}{R_u(l)} < \overset{j}{W_u(l)}$

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

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

$$\overset{k}{W_{u'}(l)} < \overset{i}{R_u(l)} < \overset{j}{W_{u''}(l)}$$

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

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 $\mathit{event}(i) = R_u(l)$, $\mathit{event}(j) = R_u(l')$, and $\mathit{ts}(i) \neq \mathit{ts}(j)$. 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 $\mathit{ts}(i)$ and $\mathit{ts}(j)$. We have that $i' < i$ and $j' < j$, and $\mathit{event}(i') = W_{u'}(l)$, $\mathit{event}(j') = W_{u''}(l')$.

Case 1. $l = l'$. Then it must be that:

$$\overset{i'}{W_{u'}(l)} < \overset{i}{R_u(l)} < \overset{j'}{W_{u''}(l)} < \overset{j}{R_u(l)}$$

We have $\mathit{thread}(u) \neq \mathit{thread}(u'')$ 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 $\mathit{ts}(i) < \mathit{ts}(j)$.

Case 2a. $\overset{i'}{W_{u'}(l)} < \overset{i}{R_u(l)} < \overset{j'}{W_{u''}(l')} < \overset{j}{R_u(l')}$

The unit of work u'' must write l as well by Assumption 1. Let k be the index of such a write, $\mathit{event}(k) = W_{u''}(l)$. 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 $\mathit{thread}(u) \neq \mathit{thread}(u')$, 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. $\overset{i'}{W_{u'}(l)} < \overset{j'}{W_{u''}(l')} < \overset{i}{R_u(l)} < \overset{j}{R_u(l')}$
or
$\overset{j'}{W_{u''}(l')} < \overset{i'}{W_{u'}(l)} < \overset{i}{R_u(l)} < \overset{j}{R_u(l')}$

Let k be an index such that $\mathit{event}(k) = W_{u''}(l)$. Since $\mathit{ts}(i') = \mathit{ts}(i) < \mathit{ts}(j) = \mathit{ts}(j') = \mathit{ts}(k)$, then it must be that $k > i'$, because otherwise that would contradict Lemma 1. We also have that $k > i$ because i' is the index responsible for the value read at i. We have that $\mathit{thread}(u) \neq \mathit{thread}(u'')$, 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 $\mathit{ts}(i) > \mathit{ts}(j)$. The unit of work u' must write l' as well by Assumption 1. Let k be the index of such a write, $\mathit{event}(k) = W_{u'}(l')$. Since $\mathit{ts}(u') = \mathit{ts}(i) > \mathit{ts}(j) = \mathit{ts}(u'')$, 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 $\mathit{thread}(u) \neq \mathit{thread}(u')$, 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 $\mathit{ts}(i) = \mathit{ts}(j)$.

$$\overset{j}{W_{u'}(l)}\quad \overset{i}{R_u(l)}\quad \overset{k}{W_{u'}(l)}\qquad \text{(scenario 3)}$$

Part 1

$$\overset{j}{W_u(l)}\quad \overset{k}{W_{u'}(l)}\quad \overset{i}{R_u(l)}\qquad \text{(scenario 4)}$$

Part 2 - Subcase 1a

$$\overset{i}{R_u(l)}\quad \overset{j}{W_{u''}(l)}\quad \overset{i'}{W_u(l)}\qquad \text{(scenario 1)}$$

Part 2 - Case 2

$$\overset{i}{R_u(l)}\quad \overset{j'}{W_{u''}(l)}\quad \overset{j}{R_u(l)}\qquad \text{(scenario 2)}$$

Part 3 - Case 1

$$\overset{i}{R_u(l)}\quad \overset{k}{W_{u''}(l)}\quad \overset{j'}{W_{u''}(l')}\quad \overset{j}{R_u(l')}\qquad \text{(scenario 9)}$$
$$\overset{i}{R_u(l)}\quad \overset{j'}{W_{u''}(l')}\quad \overset{k}{W_{u''}(l)}\quad \overset{j}{R_u(l')}\qquad \text{(scenario 9)}$$
$$\overset{i}{R_u(l)}\quad \overset{j'}{W_{u''}(l')}\quad \overset{j}{R_u(l')}\quad \overset{k}{W_{u''}(l)}\qquad \text{(scenario 10)}$$

Part 3 - Case 2a

$$\overset{j'}{W_{u''}(l')}\quad \overset{i}{R_u(l)}\quad \overset{k}{W_{u''}(l)}\quad \overset{j}{R_u(l')}\qquad \text{(scenario 11)}$$
$$\overset{j'}{W_{u''}(l')}\quad \overset{i}{R_u(l)}\quad \overset{j}{R_u(l')}\quad \overset{k}{W_{u''}(l)}\qquad \text{(scenario 8)}$$

Part 3 - Case 2b

$$\overset{i'}{W_{u'}(l)}\quad \overset{i}{R_u(l)}\quad \overset{j}{R_u(l')}\quad \overset{k}{W_{u'}(l')}\qquad \text{(scenario 8)}$$

Part 3 - Case 3

Table 1 - Proof of Lemma 2