Transforming Threads into Actors: Learning Concurrency Structure from Execution Traces

Gul Agha(✉) and Karl Palmskog

University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{agha,palmskog}@illinois.edu

Abstract. The threads and shared memory model is still the most commonly used programming model. However, programs written using threads interacting with shared memory model are notoriously bug-prone and hard to comprehend. An important reason for this lack of comprehensibility is thread based programs obscure the natural structure of concurrency in a distributed world: *actors* executing autonomously with their own internal logic and interacting at arms length with each other. While actors encapsulate their internal state, enabling consistency invariants of the data structures to be maintained locally, thread-based programs use control-centric synchronization primitives (such as locks) that are divorced from the concurrency structure of a program. Making the concurrency structure explicit provides useful documentation for developers. Moreover, it may be useful for refactoring thread-based object-oriented programs into an actor-oriented programs based on message-passing and isolated state. We present a novel algorithm based on Bayesian inference that automatically infers the concurrency structure of programs from their traces. The concurrency structure is inferred as consistency invariants of program data, and expressed in terms of annotations on data fields and method parameters. We illustrate our algorithm on Java programs using type annotations in Java classes and suggest how such annotations may be useful.

Keywords: Concurrency · Actors · Shared memory · Threads · Java
Dynamic analysis · Bayesian inference

1 Introduction

A natural way to model the world is as a collection of actors that are autonomous and concurrent [25]. The notion of actors has been developed as a programming model [1,3] and given a formal semantics [2]. An actor encapsulates (isolates) its local state; other actors may access an actor's data only through an interface defined by the latter. Such encapsulation, or isolation of data, enables us to guarantee the consistency of the data that is owned by an actor.

An alternative for concurrency is independent threads of control manipulating shared memory. As Lee points out, a fundamental problem with threads is

M. Lohstroh et al. (Eds.): Lee Festschrift, LNCS 10760, pp. 16–37, 2018.
https://doi.org/10.1007/978-3-319-95246-8_2

the unrestricted nondeterminism of thread interleavings in the absence of synchronization primitives [30]. In addition to being difficult to understand and maintain, this makes thread-based programs notoriously prone to bugs such as data races, deadlocks, and atomicity violations [36].

In practical terms, an important reason for concurrency-related bugs in thread-based programs is that control-centric synchronization primitives that enable atomic actions with respect to other threads, e.g., *locks*, are divorced from *consistency invariants* relating data structures in programs.

Consider an object representing a concurrently accessed *list*. This list object may have an array field and an integer field, where the integer field indicates how much of the array field is in use. In other words, there is an *invariant* which relates the array and integer fields. *Locks* must prevent the two fields from being modified concurrently by different threads. Otherwise, the program could exhibit a *"high-level race"* [4,9], where the list object reaches an inconsistent state. Observe that in a thread-based program, there is no explicit connection between the invariant and the code that uses locks to preserve the invariant against concurrent modification.

In order to make the *concurrency structure* in a program perspicuous, we need to make its consistency invariants explicit. Making the concurrency structure of programs with threads and shared memory explicit is desirable for several reasons. For example, if a certain consistency invariant is documented, a programmer can use this knowledge to avoid unintended atomicity violations when calling methods in existing classes and when adding new methods; both may require adding synchronization for accessing fields. Moreover, as we argue in this paper, knowledge of the concurrency structure of a multi-threaded object-oriented program can be used to *transform* this program to use the actor model, i.e., to introduce message-passing interfaces between active program components to isolate their state [1,29,31].

One way to express consistency invariants is to provide annotations on data fields and method parameters. Since we are mainly interested in the *existence* of invariants between fields, and not their exact formulation in some program logic such as JML [13], only basic annotation facilities are required. An example of such annotations are those provided by Java 8 [43]. In earlier work, Vaziri et al. provided the syntax and semantics for a set of annotations which represent consistency invariants, and proved their soundness in a minimal calculus for a Java-like language [17].

Unfortunately, manually adding consistency invariant annotations is time-consuming and error-prone. Annotating a legacy program requires understanding the program through its use of control-centric primitives. Even for relatively small and simple programs of a few hundred lines, the conversion process can take several hours. The manually produced annotations can also be problematic in two ways: first, unrelated fields may be connected by annotations; and second, related fields may not be connected. The first type of erroneous annotations underestimate the permitted degree of concurrency; the second type is consistent with unintended executions due to high-level data races. Several techniques

based on either *static* or *dynamic* program analysis to infer annotations related to consistency invariants have been proposed [15,27,34,37]. However, these techniques have serious limitations–both in the kind of data-centric primitives they can infer, and in the precision and stability of their results.

In this paper, we present a novel machine learning algorithm that automatically infers consistency invariant annotations for concurrent programs using the threads and shared memory model from program execution traces. The algorithm, called Bayesian Annotation Inference Technique (BAIT), is based on *Bayesian inference* [44]. BAIT achieves robustness against intermittent deviations from normal behavior in a trace by weighing such occurrences against a preponderance of contrary evidence and correspondingly devaluing their impact. BAIT performs its analysis *on-the-fly* and scales to large programs and long executions. BAIT improves the accuracy of its results as the number of observations grows by taking into account the *distance* (in terms of basic operations) between two related observations, thus distinguishing between unrelated computation phases. Finally, we discuss how such annotations may be facilitate transformation of threaded programs analyzed into actor-based programs.

2 Concurrency Structure Annotations and Actors

To concretize our discussion of concurrency structure, annotation inference, and actors, we consider Java programs. We describe our concurrency structure annotations for these programs, and how they related to program behavior using a running example. We then outline how annotations can be used to transform the example program to use actors.

2.1 Syntax and Semantics of Annotations

To capture consistency invariants, we adapt core constructs from the calculus of Dolby et al. [17] to the syntax of Java 8 type annotations. We consider three kinds of annotations: *atomic sets*, *aliases*, and *unitfors*. An atomic set is a group of fields inside an object that are connected by a consistency invariant; objects can contain multiple, but disjoint atomic sets. An alias extends atomic sets beyond object boundaries–an alias merges the atomic set containing a field with an atomic set in the object that is the field's value. A unitfor intuitively merges atomic sets of objects passed as parameters to a method with atomic sets in the callee object, but only for the duration of the method call.

Figure 1 shows Java code with our annotations. The `@AtomicSets` annotation on line 1 declares an atomic set `L`, and the `@Atomic("L")` annotations of the field declarations for `size` and `elements` and these fields to `L`. The class `List` corresponds to the example mentioned in Sect. 1: the value of a list's `size` integer field must equal the number of elements in the `elements` array actually used to store list entries, so the fields `size` and `elements` form an atomic set. Each `List` object has its own atomic set `L`. Recall that atomic sets express the *existence* of

```
@AtomicSets({"L"})
class List {
  @Atomic("L") int size;
  @Atomic("L") Object[] elements;

  public int size() {
    return size;
  }
  public Object get(int index) {
    if (0 <= index && index < size)
      return elements[index];
    else
      return null;
  }
  public void
    addAll(@UnitFor("L") List o) {
    this.size = this.size + o.size;
    /* ... */
  }
  /* ... */
}
```

```
@AtomicSets({"U"})
class DownloadManager {
  @Atomic("U") @Alias("L") List urls;

  public boolean hasNextURL() {
    return urls.size() > 0;
  }
  public URL getNextURL() {
    if (urls.size() == 0)
      return null;

    URL url = (URL) urls.get(0);
    urls.remove(0);
    announceStartInGUI(url);
    return url;
  }
  /* ... */
}
```

Fig. 1. Example annotated Java classes.

consistency invariants, without requiring an explicit expression such invariant, e.g., `size < elements.length`, for the class `List`.

Semantically, an atomic set is associated with one or more *units of work*. A unit of work is a method that preserves the consistency of its associated atomic sets when executed sequentially. Thus, atomic sets can ensure the application's consistency by inserting synchronization operations that guarantee the sequential execution of all units of work. By default, all non-private methods of a class are units of work for all atomic sets declared in the class or any of its subclasses. Like field declarations, atomic sets use classes as scopes, but are instance specific at runtime. Consider the methods `get(int)` and `addAll(List)` from Fig. 1. Each method is (implicitly) a unit of work for the atomic set `L` of its receiver `List` object. Hence, two threads, t_1 and t_2, that concurrently invoke `get(int)` and `addAll(List)` on a `List` l cannot interleave when accessing l's field: either t_1 executes `get(int)` first, or t_2 executes `addAll(List)` first. The interleaved case where t_2 has updated l`.size` but not l`.elements`, which causes t_1 to violate the array bounds cannot occur.

For aliases, consider the `DownloadManager` class from Fig. 1. The alias annotation `@Alias("L")` of the `urls` field declaration combines the atomic set `L` in `List` with the atomic set `U`. Hence, the method `getNextURL()` is a unit of work for this combined atomic set; its access to the `urls` list cannot be interleaved, which guarantees that no other thread can empty the list between the invocations of `urls.get(0)` and `urls.remove(0)`.

The `unitfor` annotation allows methods to be declared as units of work for atomic sets in the method's parameters. For example, the method `addAll(List)` is not only a unit of work for the atomic set `L` of its receiver `List` object but also for the atomic set `L` of its argument. Hence, if two threads, t_1 and t_2, concurrently invoke `get(int)` on a `List` l and pass the same l as the argument to `addAll(List)`, they still cannot interleave when accessing l's field.

The program in Fig. 2 illustrates how the classes in Fig. 1 can be used to implement concurrent downloading of a collection of files given as URLs. Note that the program uses control-centric synchronization in the form of Java monitors. While this particular use of monitors gives rise to program behavior consistent with the meaning of the concurrency structure annotations in the `List` and `DownloadManager` classes, other uses can easily cause atomicity violations. We suggest that another option is to refactor the program to use actors and message-passing rather than explicit threads and synchronization, as described below.

```
class DownloadThread extends Thread {
  DownloadManager manager;

  public void run() {
    while (true) {
      URL url;
      synchronized(manager) { // atomic access to manager
        if (!manager.hasNextURL()) break;
        url = manager.getNextURL();
      }
      download(url); // blocks while waiting for data
    }
  }
  /* ... */
}

public class Download {
  public static void main(String[] args) {
    // create manager and thread objects
    DownloadManager manager = new DownloadManager();
    for (int i = 0; i < 31; i++) {
      manager.addURL(new URL("http://www.example.com/f" + i));
    }
    DownloadThread t1 = new DownloadThread(manager);
    DownloadThread t2 = new DownloadThread(manager);
    t1.start();
    t2.start();
  }
}
```

Fig. 2. Example download program that uses threads to manage multiple network connections. Threads share a single manager that maintains the list of URLs to download. The program uses control-centric synchronization (Java monitors).

2.2 From Annotations and Threads to Actors

The aliased atomic sets in `List` and `DownloadManager` in Fig. 1 suggest that we can encapsulate as a single actor one instance of the latter containing an instance of the former. Then, instead of synchronizing on a `DownloadManager` instance, which would require assuming shared memory, we can simply use message passing to retrieve URLs and rely on actors having a single locus of control.

To enable wrapping objects into actors and passing (immutable) actor names instead of in-memory object references, we also have to change the program to rely on interfaces rather than classes directly in the code. Figure 3 shows

two classes in the resulting program. We use the syntax **new actor** to indicate actor-wrapped objects.

```
class DownloadThread extends Thread implements IDownloadThread {
  IDownloadManager manager;

  public void run() {
    URL url;
    // message-passing semantics
    while((url = this.manager.getNextURL()) != null) {
      download(url); // blocks while waiting for data
    }
  }
  /* ... */
}

public class Download {
  public static void main(String[] args) {
    // actor initializations
    IDownloadManager manager = new actor DownloadManager();
    for (int i = 0; i < 31; i++) { // message-passing semantics
      manager.addURL(new URL("http://www.example.com/f" + i));
    }
    IDownloadThread t1 = new actor DownloadThread(manager);
    IDownloadThread t2 = new actor DownloadThread(manager);
    t1.start();
    t2.start();
  }
}
```

Fig. 3. Actor program involving classes from Figs. 1 and 2.

Note that in order to execute the program in Fig. 3, the runtime system must perform actor initialization on **new actor** assignments and convert method invocations on actor names to message passing. However, not all programs can be straightforwardly converted in this way. In particular, to preserve program semantics, data passed through a message passing interface must have the same meaning to the sender and the receiver.

Consider the example where the receiver resides in a different runtime environment, memory references from the sender will not be valid: a message m sent from actor A to actor B at runtime may contain references to objects at A's location which do not exist at B's location. What is required is that the data passed in messages be immutable, e.g., consist of only actor names and constants. In the example actor program, only actor references (`manager`) and immutable URL objects are passed in messages. In the case of a Java Virtual Machine (JVM) based actor framework such as Akka [32], A may live in a different JVM than B, so that some object references in m do not refer to meaningful memory locations in the JVM of B. Even when actors live in the same JVM, Akka developers recommend messages be made *immutable* [33], i.e., that their contents are passed as (unchangeable) values. This was the case in the actor program in Fig. 3: the only objects passed in messages have class URL, and all its instances are by nature completely immutable.

Nevertheless, passing whole mutable objects (rather than references) in actor messages can be consistent with the behavior of the original object-oriented program in some situations. For example, if actors other than the receiver do not interact with the object at all after the message with the object has been sent, different actors will never have an inconsistent view of the object. More generally, if actors pass *ownership* of objects in messages [8], behavior is preserved in a distributed environment. Weaker guarantees than ownership passing may sometimes be acceptable, such as when actors promise not to call methods that mutate a received object's state [39]. For example, the *Pony* object-oriented actor language has a *type system* which can account for many of these situations and guarantee expected behavior in distributed program runtime settings [12]; however, the programmer must add such type annotations manually.

We believe that both static and dynamic inference can assist in inferring properties such as immutability and ownership passing to establish preservation of behavior of actorized programs with their original purely object-oriented behavior [5,39,40,45]. If safe message-passing behavior cannot be established for some methods automatically by inference, one option is to make the actor decomposition more *coarse-grained*, i.e., let fewer objects be wrapped by actors in the program translation. This results in less concurrency, and less flexibility in distributing actors at runtime to different locations, but does not require complex refactoring of the program to ensure the actor message-passing semantics preserves the original behavior.

3 Annotation Inference Example

In this section, we use an example based on the code in Sect. 2 to explain the key concepts in Bait. Suppose that we are given the classes `List` and `DownloadManager` from Fig. 1 but without the annotations. Moreover, we use the code from Fig. 2 that downloads files in parallel, and manages its network connections via threads. (Note that the synchronization in the `DownloadThread` method `run()` makes calls to the `DownloadManager` instance atomic.) We show how our algorithm infers the annotations during an execution of the program.

Bait observes program execution as a sequence of concurrency-related events. One such sequence is partially displayed in Fig. 4. Bait infers data-centric synchronization annotations based on the two following intuitions:

1. the fields of an object that a thread accesses together, without interleaving, likely belong to the same atomic set; and
2. groups of objects that a thread accesses together are likely to be connected by aliases.

In the partial execution shown in Fig. 4, one of the downloading threads invokes `getNextURL()` to request a new URL to download from the shared manager. After ensuring that the list of pending URLs contains an entry, the manager picks and removes the first entry. The manager then announces the start of the download in the program's user interface and finally returns the value to the thread.

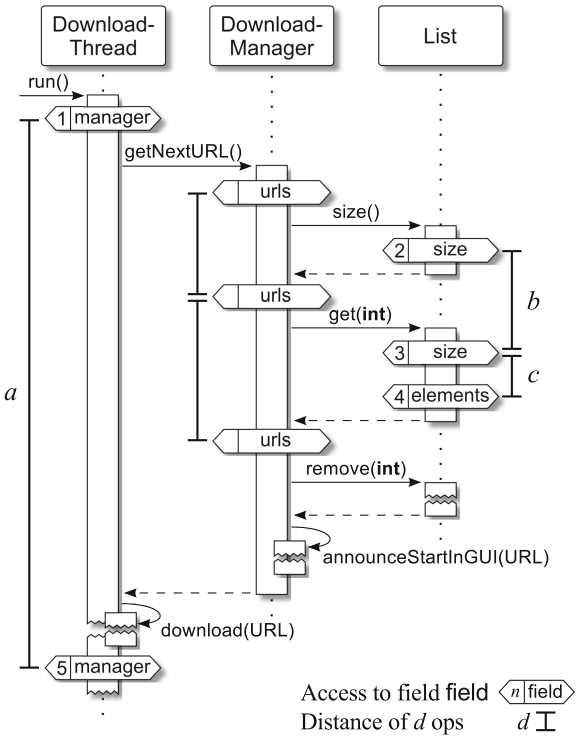

Fig. 4. Sample execution of the program from Fig. 2 used to demonstrate the basic ideas of the algorithm.

3.1 Inference of Atomic Sets

BAIT assumes that the methods of a program perform semantically meaningful operations and that the trace during an execution (mostly) represents the intended behavior of the program—for example, such a trace may be generated by running an existing integration test.

Given these assumptions, the fields of an object accessed atomically by a method in close succession are likely connected by some invariant. The set of fields that a method accesses atomically is consequently a *candidate atomic set*; the method itself is a *candidate unit of work* for this atomic set. For example, the `get(int)` method reads the fields `size` and `elements` in the same list object. In the sample execution of Fig. 4, the reads (accesses 3 and 4) happen close together and without interleaving. Thus, we have evidence that the class `List` should contain an atomic set with these two fields. Method `get(int)` is the *context* of the field accesses and thus a unit of work for this potential atomic set.

However, field accesses within a method may be far apart. For example, the two accesses to the thread object's `manager` field in the `run()` method of `DownloadThread` (1 and 5) are separated by a method call with many opera-

tions. Observing a large *distance* like a between two field accesses diminishes the likeliness of an invariant between the fields. Such an observation hence counts as evidence *against* an atomic set containing the fields. The same is true for interleaved accesses to fields by multiple threads.

The central idea of the algorithm is to use this evidence for and against atomic sets in Bayesian inference. Collecting evidence, BAIT updates its *belief* that fields belong to the same atomic set. If the belief is high enough at the end of the execution—intuitively, there was stronger evidence for an atomic set than against it—BAIT outputs corresponding **`@Atomic`** annotations.

3.2 Inference of Aliases

Since high-level semantic operations often employ low-level operations, field accesses may belong to different contexts. In Fig. 4, access 2 happens within the `size()` method, while access 3 happens within the `get(int)` method of `List`. Increasing the distance between the accesses ($b > c$) suffices to adjust the atomic set evidence in this case. However, the context that contains both accesses is no longer obvious.

The algorithm uses the lowest common ancestor in the call tree as the context for field accesses belonging to different methods. For accesses 2 and 3, e.g., this is the `getNextURL()` method. Intuitively, we observe a pair of nearby atomic accesses to `urls.size` within that context. Besides being evidence for an atomic set containing field `size`, this suggests that `getNextURL()` is a unit of work for this atomic set. Because the method accesses `size` via the field `urls`, there should be an alias from the atomic set containing `urls` to the one containing `size`.

However, aliases can remove all concurrency from the program when they include objects shared between threads. In Fig. 5, two download threads share a manager. Each thread's `run()` method is context for two nearby atomic accesses to the field `urls` in the manager object (accesses 6, 7 and 8, 9). Performing inference as above, this suggests an alias that merges the atomic set in class `DownloadThread` containing the `manager` field with the atomic set in `DownloadManager` containing the `urls` field. The alias would make the `run()` method a unit of work for the manager's atomic set that contains the `urls` field. As a consequence, the execution of the `run()` methods must be sequentialized, which would mean that only one of the two threads can be active at all, reducing performance.

BAIT mitigates the sequentialization problem by tracking which objects threads access together and weakening the belief in aliases across the boundaries of such object clusters. In our example, both threads access themselves, the manager, and the list object. Thus, the heuristic detects three clusters of objects: two that are accessed by a single thread (the thread objects themselves), and one that is accessed by both threads (the manager and the list object). Maintaining the boundaries between these clusters, the heuristic prevents aliases from `manager` to `manager.urls`, but it allows an alias from `urls` to `urls.size`.

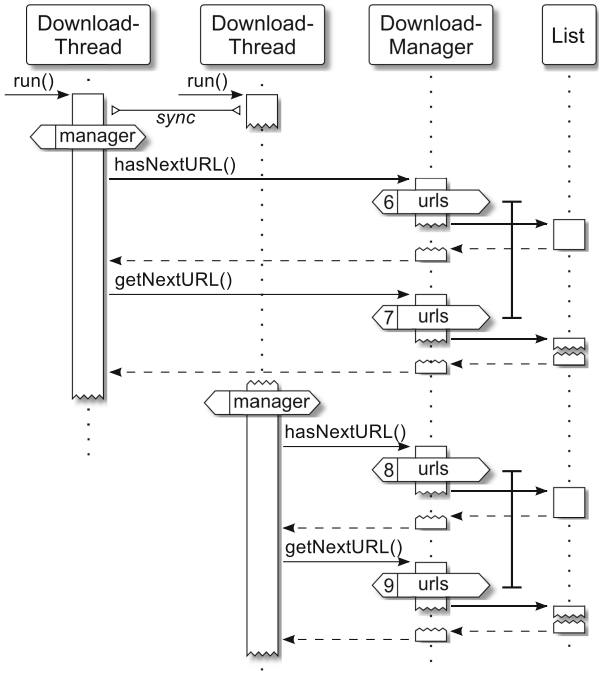

Fig. 5. Sample execution of the program in Fig. 2 that highlights a challenge in alias inference.

4 Algorithm

In this section, we describe BAIT in detail, building upon the ideas introduced in Sect. 3.

4.1 Field Access Observations

During the execution of a workload, the algorithm records *get* and *put* operations on the fields of each object. These observations are captured in the scope of a method call for a thread. From two consecutive observations for the *same* object, BAIT generates a *field access event* e, which is a tuple

$$(f, g, d, a) \in \mathrm{Fd} \times \mathrm{Fd} \times \mathbb{N} \times \mathrm{At}.$$

Here, Fd denotes the set of all fields in the program; f is the first field accessed, g is the second field. The distance d between the two accesses is the number of basic operations executed by the thread, such as Java byte code instructions. The entry $a \in \mathrm{At} = \{\mathrm{atomic}, \mathrm{interleaved}\}$ signals whether access to both f and g was atomic or access to g was interleaved with some other thread. To detect such interleaved accesses, BAIT relies on a separate race detection algorithm such as FastTrack [21], which is used in the implementation described in Sect. 5.

4.2 Bayesian Detection of Semantic Invariants

Using the generated field access events, the algorithm aims to determine whether there are invariants that hold between pairs of fields. Consider two fields f and g accessed in method m of a thread when executing a program on some workload. Suppose the workload generates the events $e_1, \ldots, e_n$, all related to f and g. Write H for the hypothesis that there exists a semantic invariant connecting f and g in the method, and $\neg H$ for the negated hypothesis that there is no such invariant.

Our goal is to find out to what degree the *evidence*, in the form of $e_1, \ldots, e_n$, supports the conclusion that H holds. In the Bayesian probabilistic reasoning framework [44], this degree of support is formalized as the conditional probability of H given $e_1, \ldots, e_n$, which through Bayes's formula can be written as

$$P(H|e_1, \ldots, e_n) = \frac{P(e_1, \ldots, e_n|H) \cdot P(H)}{P(e_1, \ldots, e_n)}. \tag{1}$$

Unfortunately, the right-hand side is difficult to estimate because it would require guessing the absolute probability that the events $e_1, \ldots, e_n$ occur in a program. For estimation, it is more convenient to use relative values such as the so-called odds and likelihood ratios. Intuitively, the likelihood ratio expresses how many times more likely an event is when the hypothesis is true versus when the hypothesis is false. Thus, we divide the left-hand side of Eq. 1 with its complementary form, yielding

$$\frac{P(H|e_1, \ldots, e_n)}{P(\neg H|e_1, \ldots, e_n)} = \frac{P(e_1, \ldots, e_n|H)}{P(e_1, \ldots, e_n|\neg H)} \cdot \frac{P(H)}{P(\neg H)}.$$

What the equation says is that our revised belief in H, when presented with $e_1, \ldots, e_n$, is equal to the ratio of the chances of observing $e_1, \ldots, e_n$ under H and $\neg H$, times our initial belief in H. We call the revised belief *posterior odds*, the ratio of the chances of making observations the *likelihood ratio*, and our initial belief the *prior odds*. More compactly, then, we write the equation as

$$O(H|e_1, \ldots, e_n) = L(e_1, \ldots, e_n|H) \cdot O(H). \tag{2}$$

These quantities are easier to estimate than probabilities, yet must be recomputed from scratch every time new evidence is added. However, if $e_1, \ldots, e_n$ are *conditionally independent* given H, an assumption discussed in Sect. 4.3, we have

$$P(e_1, \ldots, e_n|H) = \prod_{k=1}^{n} P(e_k|H),$$

and similarly for $\neg H$, which together with Eq. 2 gives

$$O(H|e_1, \ldots, e_n) = O(H) \cdot \prod_{k=1}^{n} L(e_k|H).$$

This equation suggests that recursive, on-the-fly computation of odds is possible, as becomes clear when adding one more piece of evidence e_{n+1}, yielding

$$O(H|e_1, \ldots, e_n, e_{n+1}) = L(e_{n+1}|H) \cdot O(H|e_1, \ldots, e_n).$$

We set $O(H) = 1$, that is, we assume that H and $\neg H$ are initially equally likely. We have thus reduced the problem of obtaining the degree of support for H to computing $L(e|H)$, given the data from e.

4.3 Conditional Independence of Events

Conditional independence means that knowledge of H, or $\neg H$, makes evidence up to that point irrelevant with respect to future evidence. Equivalently, under conditional independence, the hypothesis influences the evidence directly, without systematic interference from external factors. However, in a run of the algorithm directly on the JVM, the evidence produced can clearly be skewed through systematic influence from the chosen workload and the scheduler. While a workload is simple to revise, controlling thread schedules is difficult for programs running on the JVM. JPF provides a virtual machine implemented on top of the regular JVM that enables full control over nondeterminism such as scheduling points. Hence, running the algorithm on JPF with random scheduling can rule out influence by the scheduler.

Another way to address this problem is to refine the (coarse-grained) hypothesis space that either H or $\neg H$ holds into multi-valued variables [44]. This leads to a considerably more complicated mapping of evidence to likelihoods ratios. Instead of taking this route, we argue that the influence of external factors can be minimized by running Bait on workloads with sufficient code coverage for long enough to exhibit all critical interleavings, using JPF where feasible.

Although Bait can falsely conclude that two fields are related by an invariant (and thus include them in an atomic set or add an alias) when they are not, the resulting behavior is still safe. However, performance may suffer because of such an error, due to increased overhead from synchronization and reduction of concurrency.

4.4 Estimation of Likelihood Ratios

Suppose the field access event e reports we have a distance d between atomic accesses of f and g. Intuitively, the likelihood ratio $L(e|H)$ we assign based on e should have the following properties:

1. As d decreases, $L(e|H)$ must increase, but only up to some point, after which it becomes a flat maximum value; even if atomic accesses of f and g happen in close proximity, it is not conclusive that H holds.
2. As d increases, $L(e|H)$ must decrease, but only to some minimum value greater than zero; one observation should not make it impossible to conclude that H holds.

BAIT therefore uses a *logistic function* $\ell(d)$, as shown in Fig. 6, to map field access events to likelihood ratios. For example, accesses 2, 3, and 4 in Fig. 4 occur in close succession. We interpret this as evidence that it is more likely than not that an invariant connects the fields `size` and `elements`. Hence, we assign the distances b and c, with $b > c$, likelihood ratios $\ell(c) > \ell(b) > 1$. In contrast, the large distance a diminishes our belief that an invariant connects the two accesses to the `manager` field. Thus, we set $1 > \ell(a) > 0$. We leave the exact parameters of the logistic curve—its steepness and minimum and maximum likelihood ratios—to be determined during an implementation of the algorithm.

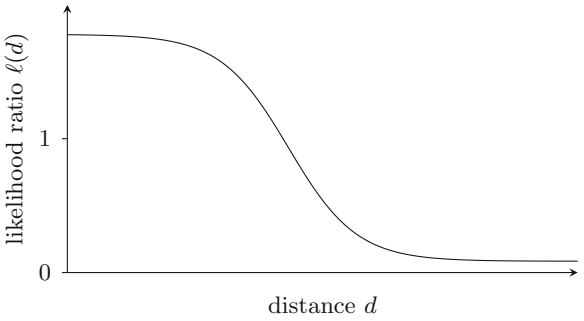

Fig. 6. Logistic curve for mapping atomic-access distances to likelihood ratios.

However, distance is not the only criterion for estimating the likelihood ratio. Suppose that e reports interleaved access. We then disregard the distance and set $L(e|H)$ to a real number p ("penalty") close to zero. This reflects that, intuitively, our belief in an invariant goes down significantly after witnessing interleaving, while not making it impossible to infer the invariant's existence later on, through overwhelming atomic access. BAIT is thus robust against sporadic errors like very rare data races. We again leave the precise value of p to an implementation.

In summary, given a field access event $e = (f, g, d, a)$, we define the estimated likelihood ratio for e as

$$\ell(d, a) = \begin{cases} \ell(d) & \text{if } a = \text{atomic;} \\ p & \text{if } a = \text{interleaved.} \end{cases}$$

4.5 Belief Configurations

We can now define how BAIT stores odds of invariants and uses likelihood ratios to update these odds in the course of workload execution.

Odds are stored in *affinity matrices*. An affinity matrix A is a symmetric map from pairs of fields (f, g) to real numbers. Symmetric means that the value assigned to (f, g) equals the one assigned to (g, f). Setting x as the value of (f, g), written $A[(f, g) \mapsto x]$, maintains the symmetry: after the update, it is $A(g, f) = x$.

Belief configurations describe the algorithm's state. A belief configuration B contains an affinity matrix A_m for every method m. Recall that an access event for a thread t in method m is a tuple consisting of two fields $f, g \in \mathrm{Fd}$, a distance $d \in \mathbb{N}$, and an atomicity indicator $a \in \mathrm{At}$. The transition function for belief configurations

$$\delta_{t,m} : \mathrm{Config} \times \mathrm{Fd} \times \mathrm{Fd} \times \mathbb{N} \times \mathrm{At} \to \mathrm{Config}$$

is now defined as $\delta_{t,m}\big(B, (f, g, d, a)\big) = B[m \mapsto A'_m]$ with

$$A'_m = A_m\big[(f, g) \mapsto \ell(d, a) \cdot A_m(f, g)\big]. \tag{3}$$

For all methods m, define an initial affinity matrix A_m^{init} such that $A_m^{\mathrm{init}}(f, g) = 1$ for all $(f, g) \in \mathrm{Fd} \times \mathrm{Fd}$ and an initial belief configuration B^{init} with $B^{\mathrm{init}}(m) = A_m^{\mathrm{init}}$. Then, if the events $e_1, \ldots, e_n$ are generated in t for m, the algorithm computes the final belief configuration

$$\delta_{t,m}(\cdots \delta_{t,m}(B^{\mathrm{init}}, e_1) \cdots, e_n).$$

4.6 Inference of Aliases and Unitfors

Inference of aliases and unitfors is done at the same time as inference of atomic sets, and in a similar way, but with several important differences.

Suppose we observe an atomic access of the field g after an access of f in the method m. Within m, the object that contains f and g may be known by a source code identifier, that is, by a field or parameter name n. For example, in Fig. 4, the field access event generated for the accesses 2 and 3 occurs in method `getNextURL()`. Within that method, the list object that contains the accessed `size` field is known by the field name `url`. Hence the method observes the accesses in the context of this name, as `urls.size`; and more generally, m observes the accesses of f and g as $n.f$ and $n.g$.

Such an observation indicates that m performs multiple operations on another object (the list in our example). As before, if the distance d between the accesses $n.f$ and $n.g$ is small, then these operations likely maintain an invariant. Therefore, they should be atomic, which means that an atomic set containing n should be extended—by an alias—to also contain $n.f$ and $n.g$. Translated to our example, the close accesses to `urls.size` count as evidence for an alias that merges the manager object's atomic set containing `urls` with the list object's atomic set containing `size`.

In summary, to infer aliases and units of work, we associate with each identifier n an affinity matrix A_n, and update this matrix with the likelihood ratio $\ell(d)$, penalizing interleaved accesses as for atomic sets above. Then, most straightforwardly, if $A_n(f, g) > 1$ for A_n in the final configuration, this suggests an alias from the inferred atomic set of n—should n be a field name—to the inferred atomic set of f and g. Should n be a parameter of m, then this suggests declaring m a unit of work for the atomic set of f and g in n.

Preventing Global Locks. Without further adjustments, inferring aliases this way can lead to undesirable global locks, as shown in Fig. 5: if an alias merges an atomic set in a thread object with an atomic set S in an object shared between threads, then the thread's methods become units of work for S. Consequently, only one thread object can execute at a time, making (this part of) the program sequential.

We apply the following heuristic to detect this situation and lower the respective alias beliefs. Whenever a thread t accesses a field in object o, we record t as the owner of o. Using this data, we maintain an *alias factor* α for objects. Consider the situation in Fig. 5, just after the left thread t_l's call to `getNextURL()` has returned. At this point, t_l owns itself, the manager object, and the list object. When the right thread t_r accesses its local `manager` field just after that, BAIT detects that t_r owns the object that contains the accessed `manager` field (itself), but another thread owns the object that is the field's *value* (t_l owns the manager object). Therefore, the thread object t_r and the manager object appear to belong to two different clusters in the object graph upon which different threads operate concurrently. Merging these clusters with an alias would remove the concurrency. Therefore, we set a fixed alias factor α in the range $(0, 1)$ for the manager object (the field's value). Otherwise, if t_r was the owner of itself *and* the manager object, we set α based on the recorded (same-thread) distance between the accesses, which can result both in lowering or raising belief in an alias.

Given an atomic field access event, we use the computed alias factor α for the field-containing object as weight when updating an alias affinity matrix. Adapting Eq. 3, the updated affinity matrix A'_n for the name n of o is thus computed as

$$A'_n = A_n\big[(f, g) \mapsto \alpha \cdot \ell(d, a) \cdot A_n(f, g)\big].$$

In the example shown in Fig. 5, the alias factor $\alpha < 1$ for the manager object prevents the small distance between the observed accesses of `manager.urls` ($6, 7$ and $8, 9$) in the `run()` methods from increasing the odds of the problematic alias from `DownloadThread.manager` to `DownloadManager.urls`.

A slight modification of the heuristic is necessary to account for clusters consisting of more than two objects. In its current form, the heuristic detects a different owner thread for the first accessed object o of a cluster, and the same owner for the second object v, say, accessed via field f in o. However, the access of f establishes the current thread as the owner of o. Thus, when accessing a third object w via the field g in o, the heuristic would detect different owners again, discouraging an alias even though the previous thread operated on o, v, and w. BAIT solves this problem by not only recording the current owning thread t_o for each object, but also the previous (distinct) owning thread t'_o. Different clusters are detected only if $t_o \neq t_v$ and $t'_o \neq t_v$. Thus, for the access of w we have $t'_o = t_w$ and correctly associate w with o and v.

4.7 Atomic Set, Alias and Unitfor Formation

After the workload of the program has finished executing, all atomic set field affinity matrices are merged into a single matrix. From this combined matrix,

the atomic sets are extracted by using the matrix values as edge-weights on the fully-connected graph of all fields (node set Fd), removing the edges with weight less than a threshold (e.g., 1), and grouping the fields in the remaining connected components by their declaring class (accounting for inheritance). The atomic sets are added as annotations to the class hierarchy, which forms the basis for computing aliases using the alias affinity matrices. Finally, unitfors are inferred using the class hierarchy and the alias affinity matrices.

5 Implementation Concerns

Most directly, Bait can be implemented for Java programs using instrumentation at the byte code level. In an initial phase, the instrumented byte code is executed, allowing the Bait implementation to record field accesses and build and update the affinity matrices. In the next phase, the Bait implementation can use the final affinity matrices to infer and output the annotations, or even fully annotated code.

While the algorithm mandates logistic functions for mapping distances to likelihood ratios for atomic sets and aliases, an implementation may settle for an approximation of such a function, e.g., a coarse-grained piecewise approximation. Minor extensions of the algorithm may be necessary to handle realistic Java programs, which may contain arrays, synchronized blocks, and wait–notify synchronization. Earlier work may be prove pertinent for such extensions [16]. Additional optimizations are possible, such as removal of non-aliased `final` fields from atomic sets. While the algorithm requires tracking all names that a field-owning object can have, an implementation may choose to only track the last known name at runtime. We believe this would give a reasonable tradeoff between overhead and correctness. Another implementation option besides instrumentation is to implement the dynamic analysis inside a special-purpose JVM such as Java PathFinder [42].

Finally, any implementation will have to make choices regarding several parameters that can affect the inferred annotations, most prominently the parameters that define (piecewise approximations of)) logistic functions for atomic set and alias likelihoods. Such parameters may be calibrated, e.g., on simple test cases.

6 Related Work

The automatic inference of a program's concurrency semantics has been treated in the context of data race detection. There, the concurrency semantics is used to warn about violations of the likely *intended* atomicity semantics of variables.

A dynamic approach that learns the atomicity intentions for shared variables from execution traces is the *AVIO* system of Lu et al. [37,38]. In contrast, Artho et al. [4] introduce the notion of high-level data races and explicitly design their dynamic algorithm to consider races on sets of semantically related variables. The *AssetFuzzer* algorithm of Lai et al. [28] uses partial order relaxation to detect

potential, but unmanifested, violations in the execution trace. All of these methods are similar to our algorithm in that they work without user annotations. The *Atomizer* system of Flanagan and Freund [20] additionally considers *windows of vulnerability*, but requires a few source code annotations and potentially raises false alarms. The *MUVI* tool of Lu et al. [35] follows a static approach to inferring atomicity intentions at the variable level.

The static heuristic [24,46] of defining one atomic set per class that contains all non-static fields has also been proposed in the context of race detection. Targeting race detection, none of the aforementioned approaches considers aliasing information, which is essential for our use case. Huang and Milanova's static inference system for the AJ types defined by Dolby et al. [17] significantly reduces the number of annotations that a developer has to write [27]. While simplifying the use of AJ, it needs a set of foundational annotations. Hence, their and our methods complement each other: the static inference rules propagate the base annotations inferred by our analysis, yielding a complete set of annotations. Liu et al. [34] present a technique for statically inferring atomic sets based on program dependence analysis. The inferred sets are then used for finding atomic composition bugs dynamically in programs. This is a different focus compared to our algorithm, whose main aim is to provide annotations for documentation and safe execution. In addition, our algorithm also infers aliases, which are arguably harder to infer than atomic sets, least of all statically.

Dinges et al. [15,16] present a dynamic inference algorithm of data-centric concurrency annotations as described by Vaziri et al. [17]. The algorithm is based on classification of fields into atomic sets using simple set membership criteria rather than careful weighing of evidence as in Bait. Additionally, unlike Bait, the algorithm does not scale to long executions with many field accesses, and does not improve results as more evidence becomes available; in some cases, results may even become significantly worse after observing more field accesses, since previous conclusions are replaced.

Flanagan et al. [22] present a sound and complete dynamic atomicity checker for Java programs. The tool, Velodrome, takes a workload and list of methods that are assumed to be atomic as input, and outputs a list of atomicity violations. Biswas et al. [7] improve on the significant overhead introduced by Velodrome in their DoubleChecker tool, while maintaining soundness and completeness. A tentative list of atomic methods can be derived from the annotations produced by an implementation of Bait by enumerating all methods that are units of work for some atomic set.

Atomic sets take a declarative approach to synchronization. *Synchronizers* [14,23] provide a similar notion in the context of actor systems, where they constrain the message dispatch in a group of actors. The available constraints differ from atomic sets in that synchronizers can provide *temporal* atomicity—messages arrive at the same time—not the *spatial* atomicity offered by atomic sets. Synchronizers do not support transitive extensions similar to aliases in atomic sets. Moreover, expressing the non-interleaving of message sequences, which is the actor equivalent of non-interleaved access to shared data, is more

complicated. In its simplest form, such non-interleaving in messages to a single actor is expressed in terms of *local synchronization constraints* which force an ordering on messages to a given actor [26,47]. Local synchronization constraints may be used to force FIFO ordering of messages between pairs of senders or recipients, or to ensure that a two actors follow a more complex communication protocol. Synchronizers generalize this to multiple actors: by disabling a specific type of message until another has been received, synchronizers can force an ordering between messages sent to different actors. Another declarative approach to ensuring synchronization at the actor level is that of multiparty session types [10,41].

By boosting belief in the existence of an invariant after atomic access and maintaining or possibly even strengthening that belief unless witnessing interleaved access, BAIT follows the approach of *accentuating the positive* [37,48] by suppressing rarely observed Heisenbugs that violate atomicity. Non-deadlock bugs: 74 (Atomicity: 51, Order: 24, Other: 2), Deadlock bugs: 31 A study of real-world concurrency bugs [36] finds that nearly half of all errors are related to atomicity; with deadlocks ruled out, that fraction rises to nearly 70 %. While this kind of safety comes at the cost of a coarser concurrency semantics, the experiments of Weeratunge et al. [48] suggest that a low runtime overhead of 15 % can be achieved.

The problems inherent in threads and their usual synchronization primitives, such as locks and monitors, have been examined previously, e.g., by Lee [30]. Lee argues against letting programmers start with maximally nondeterministic interleaving of threads and adding just enough determinism to avoid concurrency errors. Instead, he proposes that programmers start from a deterministic model and selectively add operations for nondeterministic composition where appropriate. While the resulting executions have more coarse-grained concurrency and thus potentially worse performance, they are inherently easier to reason about due to many thread interleavings being ruled out. Lee suggests to focus on development of coordination languages rather than thread-based primitives and libraries. We believe the concurrency structure annotations we showed in this paper can be viewed as a kind of data-driven coordination language.

7 Discussion

Although the use of actor languages (e.g., through Akka [32] and Erlang [19]) has grown dramatically in recent years, threads with shared memory and control-centric primitives continue to dominate concurrent programming. Thread-based programming often obscures key properties in programs and leads programmers to introduce concurrency bugs such as atomicity violations [36]. Moreover, it is hard to scale the thread-based model–one reason actors have been used to implement large-scale applications such as Facebook chat servers and Twitter.

In this paper, we highlight consistency invariants involving class fields and method parameters which may help reduce bugs in thread-based object-oriented programs. Unfortunately, we expect that programmers in general will not manually write, document, and check such invariants when writing thread-based

programs. Our algorithm improves on the state of the art for inferring invariant annotations automatically, freeing programmers of some of the burden.

Consistency invariant annotations are useful in several ways besides giving programmers an understanding of atomicity requirements of data structures. For example, if a class contains several atomic sets of fields, and each method accesses fields from only one atomic set, this suggests decomposing the class into several classes with methods that access only the fields in the decomposed class. While such lower-level concerns are important, we can also ask high-level questions, such as whether concurrent programs can avoid dealing with threads (partially or entirely).

As discussed earlier, the actor model avoids explicit locks by introducing a unit of computation, an *actor*, that has its own state, an independent single locus of control, and communicates with others via asynchronous message passing [3]. The actor-oriented programming approach of Lee [29,31] drops the requirement for independent control and asynchrony in message passing. In particular, Lee emphasizes the difference between object-orientation and actors as one that pertains to whether communication implies *transfer of control* from the sender to the receiver. In semantic terms, Lee's notion of actor-oriented programming incorporates programming abstractions that may be built using meta-actors which can used to customize naming and scheduling [6,18]. Clavel et al. [11] give a formal semantics to reason about such systems.

According to the hierarchy of platforms presented by Lee [29], actor-oriented models are above object-oriented programs, with the latter being closer to low-level concepts such as executables and silicon chips. From this perspective, an implementation of the algorithm we presented and our suggested actor program transformations can assist in raising the abstraction level of programs, making them amenable to conversion to concurrent programs following actor-oriented design techniques. Further exploration of such techniques will be needed to improve legacy concurrent codes, facilitating their dependability and maintainability.

Acknowledgements. The authors thank Peter Dinges, Farah Hariri, and Darko Marinov. This work is supported in part by the National Science Foundation under grants NSF CCF 14-38982 and NSF CCF 16-17401, and by AFOSR/AFRL Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement FA8750-11-2-0084 for the Assured Cloud Computing at the University of Illinois at Urbana-Champaign.

References

1. Agha, G.: Concurrent object-oriented programming. Commun. ACM **33**(9), 125–141 (1990)
2. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. J. Funct. Program. **7**(1), 1–72 (1997)
3. Agha, G.A.: ACTORS - A Model of Concurrent Computation in Distributed Systems. MIT Press Series in Artificial Intelligence. MIT Press, Cambridge (1986)

4. Artho, C., Havelund, K., Biere, A.: High-level data races. In: NDDL 2003, pp. 82–93. ICEIS Press (2003)
5. Artzi, S., Quinonez, J., Kieżun, A., Ernst, M.D.: Parameter reference immutability: formal definition, inference tool, and comparison. Autom. Softw. Eng. **16**(1), 145–192 (2009)
6. Astley, M., Sturman, D.C., Agha, G.: Customizable middleware for modular distributed software. Commun. ACM **44**(5), 99–107 (2001)
7. Biswas, S., Huang, J., Sengupta, A., Bond, M.D.: Doublechecker: efficient sound and precise atomicity checking. In: PLDI 2014. ACM (2014)
8. Boyapati, C., Lee, R., Rinard, M.C.: Ownership types for safe programming: preventing data races and deadlocks. In: OOPSLA 2002, pp. 211–230. ACM (2002)
9. Burrows, M., Leino, K.R.M.: Finding stale-value errors in concurrent programs. Concurr. Pract. Exp. **16**(12), 1161–1172 (2004)
10. Charalambides, M., Dinges, P., Agha, G.A.: Parameterized, concurrent session types for asynchronous multi-actor interactions. Sci. Comput. Program. **115–116**, 100–126 (2016)
11. Clavel, M., et al.: Reflection, metalevel computation, and strategies. In: Clavel, M., et al. (eds.) All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350, pp. 419–458. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1_14
12. Clebsch, S., Drossopoulou, S., Blessing, S., McNeil, A.: Deny capabilities for safe, fast actors. In: Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, pp. 1–12. ACM, New York (2015)
13. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 472–479. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_35
14. Dinges, P., Agha, G.: Scoped synchronization constraints for large scale actor systems. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 89–103. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30829-1_7
15. Dinges, P., Charalambides, M., Agha, G.: Automated inference of atomic sets for safe concurrent execution. In: Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE 2013, pp. 1–8. ACM, New York (2013)
16. Dinges, P., Charalambides, M., Agha, G.: Automated inference of atomic sets for safe concurrent execution. Technical report, UIUC, April 2013. http://hdl.handle.net/2142/43357
17. Dolby, J., Hammer, C., Marino, D., Tip, F., Vaziri, M., Vitek, J.: A data-centric approach to synchronization. ACM TOPLAS **34**(1), 4 (2012)
18. Donkervoet, B., Agha, G.: Reflecting on aspect-oriented programming, metaprogramming, and adaptive distributed monitoring. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 246–265. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74792-5_11
19. Erlang programming language. https://www.erlang.org
20. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. Sci. Comput. Program. **71**(2), 89–109 (2008)
21. Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, pp. 121–133. ACM, New York (2009)

22. Flanagan, C., Freund, S.N., Yi, J.: Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2008, pp. 293–303. ACM, New York (2008)
23. Frølund, S., Agha, G.: A language framework for multi-object coordination. In: Nierstrasz, O.M. (ed.) ECOOP 1993. LNCS, vol. 707, pp. 346–360. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-47910-4_18
24. Hammer, C., Dolby, J., Vaziri, M., Tip, F.: Dynamic detection of atomic-set-serializability violations. In: ICSE 2008, pp. 231–240. ACM (2008)
25. Hewitt, C., Bishop, P.B., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: Nilsson, N.J. (ed.) Proceedings of the 3rd International Joint Conference on Artificial Intelligence, Standford, CA, USA, 20–23 August 1973, pp. 235–245. William Kaufmann (1973)
26. Houck, C.R., Agha, G.: HAL: a high-level actor language and its distributed implementation. In: Shin, K.G. (ed.) Proceedings of the 1992 International Conference on Parallel Processing, University of Michigan, An Arbor, Michigan, USA, 17–21 August 1992, Volume II: Software, pp. 158–165. CRC Press (1992)
27. Huang, W., Milanova, A.: Inferring AJ types for concurrent libraries. In: FOOL 2012, pp. 82–88 (2012)
28. Lai, Z., Cheung, S.C., Chan, W.K.: Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In: ICSE 2010, pp. 235–244. ACM (2010)
29. Lee, E.A.: Model-driven development-from object-oriented design to actor-oriented design. In: Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation (a.k.a. The Monterey Workshop) (2003)
30. Lee, E.A.: The problem with threads. Computer **39**(5), 33–42 (2006)
31. Lee, E.A., Liu, X., Neuendorffer, S.: Classes and inheritance in actor-oriented design. ACM Trans. Embed. Comput. Syst. **8**(4), 29:1–29:26 (2009)
32. Lightbend: Akka. https://akka.io
33. Lightbend: Akka and the Java memory model. https://doc.akka.io/docs/akka/current/general/jmm.html
34. Liu, P., Dolby, J., Zhang, C.: Finding incorrect compositions of atomicity. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 158–168. ACM, New York (2013)
35. Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R.A., Zhou, Y.: MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In: SOSP 2007, pp. 103–116. ACM (2007)
36. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ASPLOS 2008, pp. 329–339. ACM (2008)
37. Lu, S., Park, S., Zhou, Y.: Detecting concurrency bugs from the perspectives of synchronization intentions. IEEE Trans. Parallel Distrib. Syst. **23**(6), 1060–1072 (2012)
38. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: detecting atomicity violations via access-interleaving invariants. IEEE Micro **27**(1), 26–35 (2007)
39. Milanova, A., Dong, Y.: Inference and checking of object immutability. In: Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ 2016, pp. 6:1–6:12. ACM, New York (2016)

40. Negara, S., Karmani, R.K., Agha, G.A.: Inferring ownership transfer for efficient message passing. In: Cascaval, C., Yew, P. (eds.) Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, 12–16 February 2011, pp. 81–90. ACM (2011)
41. Neykova, R., Yoshida, N.: Multiparty session actors. In: Kühn, E., Pugliese, R. (eds.) COORDINATION 2014. LNCS, vol. 8459, pp. 131–146. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43376-8_9
42. Palmskog, K., Hariri, F., Marinov, D.: A case study on executing instrumented code in Java PathFinder. In: Proceedings of JPF Workshop, JPF 2015 (2015)
43. Papi, M.M., Ernst, M.D.: Compile-time type-checking for custom type qualifiers in Java. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion, OOPSLA 2007, pp. 809–810. ACM, New York (2007)
44. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann Publishers Inc., San Francisco (1988)
45. Srinivasan, S., Mycroft, A.: Kilim: isolation-typed actors for Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70592-5_6
46. Sumner, W.N., Hammer, C., Dolby, J.: Marathon: detecting atomic-set serializability violations with conflict graphs. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 161–176. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_13
47. Tomlinson, C., Kim, W., Scheevel, M., Singh, V., Will, B., Agha, G.: Rosette: an object-oriented concurrent systems architecture. SIGPLAN Not. **24**(4), 91–93 (1989)
48. Weeratunge, D., Zhang, X., Jagannathan, S.: Accentuating the positive: atomicity inference and enforcement using correct executions. In: OOPSLA 2011, pp. 19–34. ACM (2011)