Automated Inference of Atomic Sets for Safe Concurrent Execution

Gul Agha
University of Illinois at Urbana-Champaign

Joint work with Peter Dinges and Karl Palmskog

Part I: Concurrency and Synchronization

Introduction

- Concurrency is about interaction between parties:
  - by synchronous "handshakes", or
  - by asynchronous message passing
- Concurrent parties can interact:
  - directly with each other
  - indirectly via "passive" shared data store

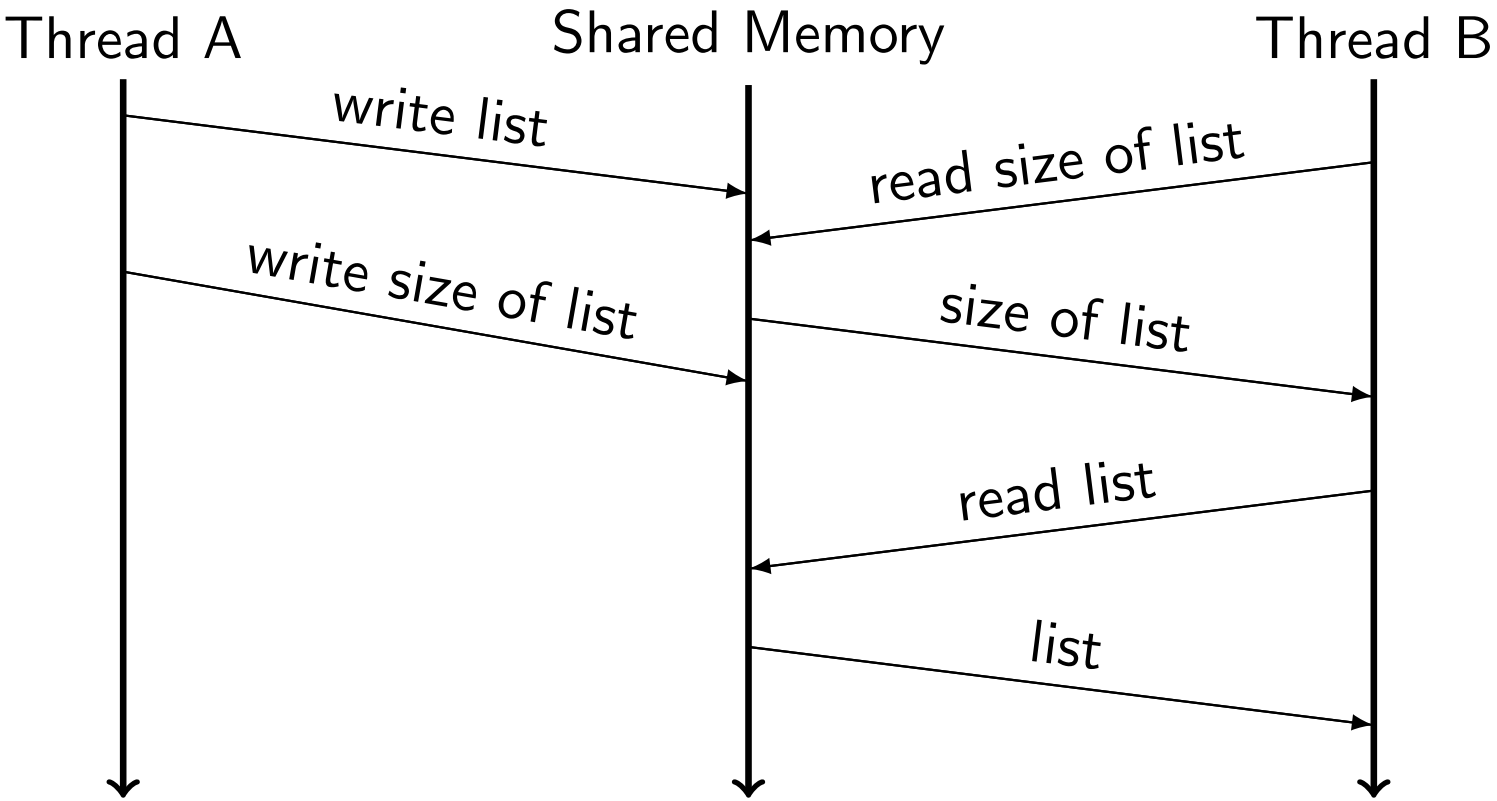
Unsafe Concurrent Behavior

- Order of interactions is not generally predetermined
- Many bugs arise from unexpected interleavings of actions:
  - data races
  - deadlocks
  - unintended order of operations

Refresher on Threads

- Multiple threads of control share a memory space
- Supported natively in most general-purpose languages
  - **Java**, C++, Python, ...
- Scheduler decisions on interleaving generally unknown
- For safety, programmers need thread **noninterference**

A High-level Data Race



Classic Control-centric Synchronization Primitives

Semaphores (Dijkstra, 1962/1963)

A data type used for synchronizing threads, associated with a counter and two operations:

- `wait`: decrements the counter if it is nonnegative; otherwise, blocks thread until this can be done
- `signal`: increments the counter by one

Classic Control-centric Synchronization Primitives

Semaphores (Dijkstra, 1962/1963)

A data type used for synchronizing threads, associated with a counter and two operations:

- `wait`: decrements the counter if it is nonnegative; otherwise, blocks thread until this can be done
- `signal`: increments the counter by one

- Still used in lower-level code such as operating systems
- Useful for resource pools with mutually exclusive access

Classic Control-centric Synchronization Primitives

Monitors (Hoare & Brinch-Hansen, 1974)

A collection of data and procedures (e.g., object) such that:

- at most one thread can be executing a procedure at any time
- procedure-calling threads are blocked if another thread is already executing
- a blocking thread is woken up when execution finishes

Classic Control-centric Synchronization Primitives

Monitors (Hoare & Brinch-Hansen, 1974)

A collection of data and procedures (e.g., object) such that:

- at most one thread can be executing a procedure at any time
- procedure-calling threads are blocked if another thread is already executing
- a blocking thread is woken up when execution finishes

- Theoretically equivalent to semaphores
- Overwhelmingly most-used synchronization primitive in real-world concurrent Java programs[1]

[1] Torres et al. (2011)

Java Monitors and Threads

```
public class SynchronizedInteger {
  private int value;
  public synchronized int get() { return value; }
  public synchronized void set(int value) { this.value = value; }
}
public class Setter extends Thread {
  private SynchronizedInteger synInt;
  /* ... */
  public void run() {
    while (true) { synInt.set((int)(Math.random()*100)); }
  }
}
public class GetSetter extends Thread {
  private SynchronizedInteger synInt;
  /* ... */
  public void run() {
    while (true) {
      synchronized(synInt) { if (synInt.get() != 0) synInt.set(0); }
    }
  }
}
```

Actors: Scalable Concurrency

Large-scale concurrent systems such as Twitter, LinkedIn, Facebook Chat are written in actor languages and frameworks.

Facebook

"[T]he actor model has worked really well for us, and we wouldn't have been able to pull that off in C++ or Java. Several of us are big fans of Python and I personally like Haskell for a lot of tasks, but the bottom line is that, while those languages are great general purpose languages, none of them were designed with the actor model at heart." –Facebook Engineering [2]

[2] https://www.facebook.com/notes/facebook-engineering/chat-stability-and-scalability/51412338919

Actors: Scalable Concurrency II

Large-scale concurrent systems such as Twitter, LinkedIn, Facebook Chat are written in actor languages and frameworks.

Twitter

"When people read about Scala, it's almost always in the context of concurrency. Concurrency can be solved by a good programmer in many languages, but it's a tough problem to solve. Scala has an Actor library that is commonly used to solve concurrency problems, and it makes that problem a lot easier to solve." – Alex Payne, "How and Why Twitter Uses Scala"[3]

[3] http://blog.redfin.com/devblog/2010/05/how_and_why_twitter_uses_scala.html

Some Actor Languages and Frameworks

- Erlang: web services, telecom, Cloud Computing
- E-on-Lisp, E-on-Java: P2P systems
- SALSA (UIUC/RPI), Charm++ (UIUC): scientific computing
- Ptolemy (UCB): real-time systems
- ActorNet (UIUC): sensor networks
- Scala (EPFL; Typesafe): multicore, web, banking..
- Kilim (Cambridge): multicore and network programming
- Orleans; Asynchronous Agents Library (Microsoft): multicore programming, Cloud Computing
- DART (Google): Cloud Computing

Actor Model of Computation

An actor is an autonomous, concurrent agent which responds to messages.

- Actors operate asynchronously, potentially in parallel with each other.
- Actors do not share state
- Each actor has a unique name (address) which cannot be guessed.
- Actor names may be communicated.
- Actors interact by sending messages which are by default asynchronous (and may be delivered out-of-order).

Actor Behavior

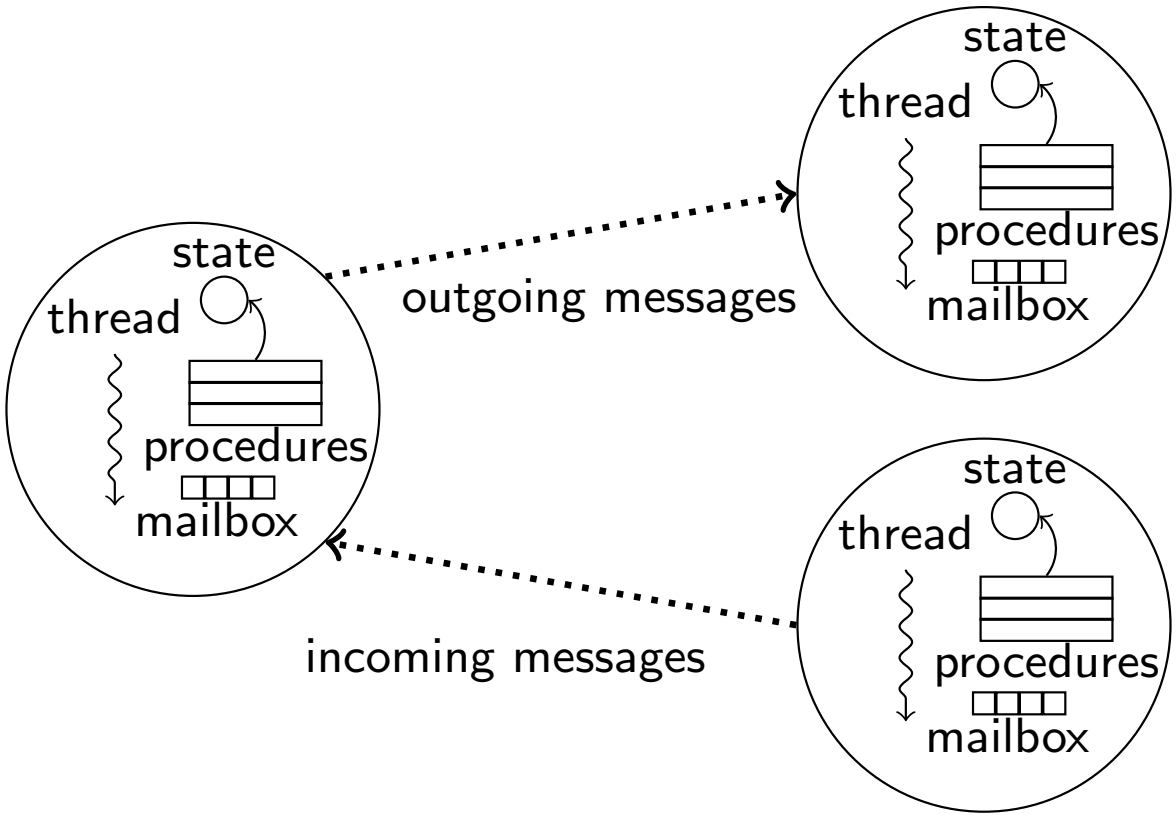
Upon receipt of a message, an actor may:

- create a new actor with a unique name (address).
- use the content of the message or perform some computation and to change state.
- send a message to another actor.

Actor Implementation in Threaded Languages

An actor may be implemented as a concurrent object. Each actor:

- has a system-wide unique name (mailbox address);
- has an independent thread of control; and
- has a message queue and processes **one message at a time**.



Actor Synchronization Constraints

- Constrain the local order of processing messages
- Function of local state and message contents
- **Delay semantics**: disabled messages are buffered (ActorFoundry[4])
- **Disabling semantics**: pattern matching (Erlang[5], Akka[6])

[4] http://osl.cs.illinois.edu/software/actor-foundry/
[5] http://www.erlang.org
[6] http://akka.io

Actor Synchronization Constraints

- Constrain the local order of processing messages
- Function of local state and message contents
- **Delay semantics**: disabled messages are buffered (ActorFoundry[4])
- **Disabling semantics**: pattern matching (Erlang[5], Akka[6])

Goals

- Eliminate problematic schedules
- Ensure harmonic interaction (coordination)

[4] http://osl.cs.illinois.edu/software/actor-foundry/
[5] http://www.erlang.org
[6] http://akka.io

Synchronization Constraints Control Concurrency

Concurrent Execution

- Many execution (message) schedules, some problematic
- Eliminate problematic schedules with constraints

Synchronization Constraints Control Concurrency

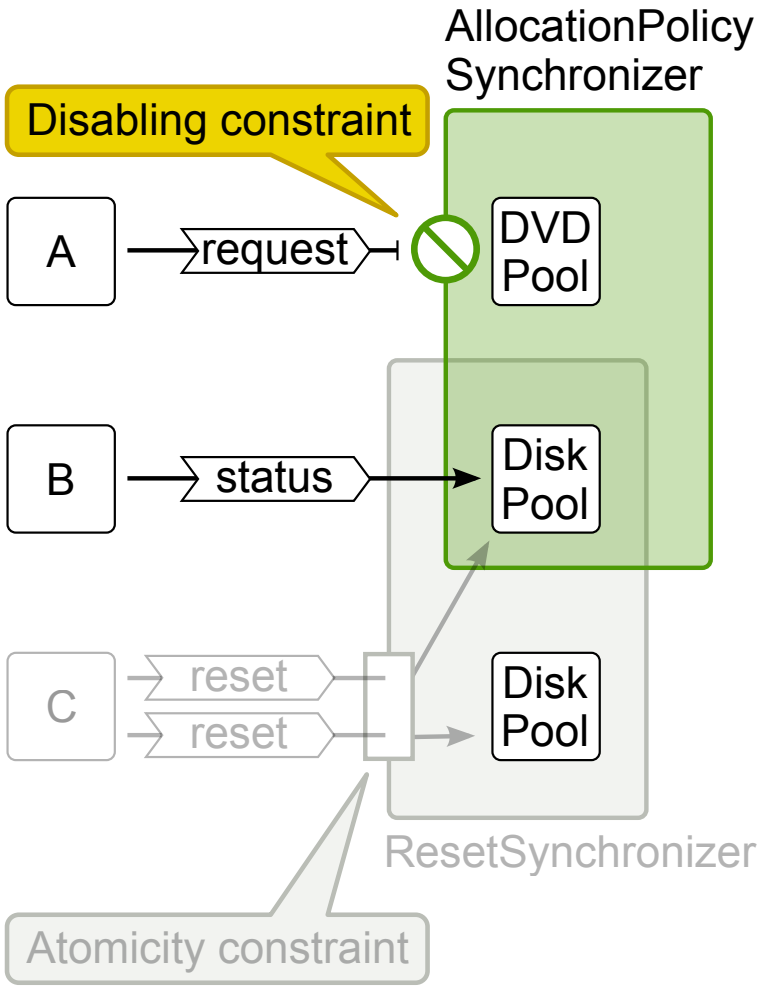
Concurrent Execution

- Many execution (message) schedules, some problematic
- Eliminate problematic schedules with constraints

Example: Dining Philosophers

- Philosophers need left and right chopstick to eat
- Potential deadlock: everyone picks left chopstick first
- Constraint: Always deliver a philosopher's pick up requests at the same time (all or nothing)

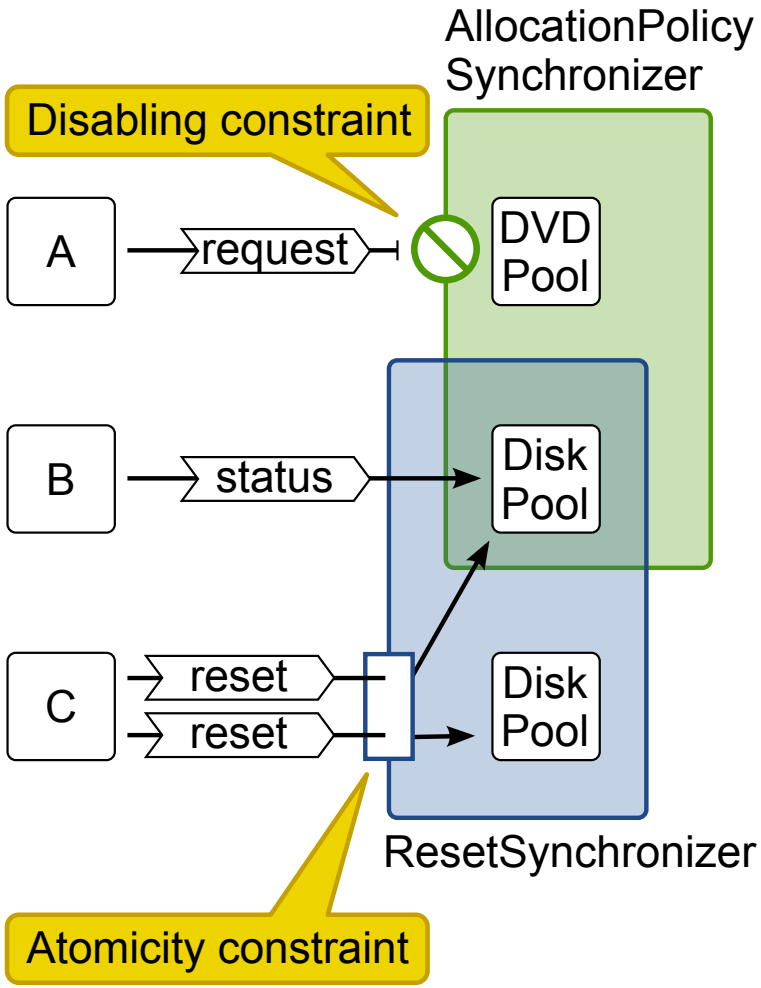
Actor Synchronizers



Synchronizers (Frølund & Agha, 1993)

- Synchronization constraints for groups of Actors
- Affect *all incoming messages* (global scope)
- **Disabling constraints** prevent handling of messages matching a pattern

Actor Synchronizers



Synchronizers (Frølund & Agha, 1993)

- Synchronization constraints for groups of Actors
- Affect *all incoming messages* (global scope)
- **Disabling constraints** prevent handling of messages matching a pattern
- **Atomicity constraints** bundle messages into indivisible sets

Example: Coordinated Resource Administrators

Scenario

- Connect DVD drives and hard disks over the network
- Total bandwidth is limited: allocate at most `max` drives

Synchronizer to coordinate resource administrators:

```
AllocationPolicy(dvds, disks, max) {
  init alloc := 0
  alloc >= max disables (dvds.request or disks.request)
  (dvds.request or disks.request) updates alloc := alloc + 1,
  (dvds.release or disks.release) updates alloc := alloc - 1
}
```

Synchronizer Use Cases and Applications

- Middleware coordination
- Web applications
- Quality of Service and multimedia applications

Data Structure Invariants

Data structures are usually associated with **invariants**, non-trivial logical properties that hold in all visible states, e.g.:

- an array that is sorted
- a priority queue with a topmost element
- a queue with FIFO ordering

Data Structure Invariants

- While data structures are updated, invariants may not hold
- In sequential code with a single control, this is not a problem
- In concurrent code, programmers must ensure **atomic access** to mutable data structures using synchronization primitives
- Simply introducing locks around all data fields is not enough, since invariants can reference many variables[7]

[7] Artho et al. (2003)

Data Structure Invariants

A study[8] of real-world concurrency bugs concluded that:

- around half of such bugs are related to atomicity
- when excluding deadlocks, atomicity bugs rise to nearly 70%
- almost all (96%) bugs required only two threads to manifest
- 2/3 of non-deadlock bugs involved access to multiple variables

[8] Lu et al. (2008)

Specifying Invariants

In object-oriented programs, data structure invariants are usually given as **class invariants**, which:

- concern class fields (data encapsulated by instances)
- are established by all constructors in the class
- are preserved by all (non-helper) instance methods in the class

Specifying Invariants in Java

Java Modeling Language[9] (JML) allows specifying as comments:

- class invariants, `//@ invariant ...`
- method preconditions, `//@ requires ..`
- method postconditions, `//@ ensures ..`
- loop invariants, `//@ loop_invariant ...`

[9] http://www.jmlspecs.org

Classic Java Wallet Class

```
public class Wallet {
  public static final int MAX_BALANCE;
  private int balance;

  /* ... */


  public int debit(int amount) {
    /* ... */
  }
}
```

Classic Java Wallet Class with JML Annotations

```
public class Wallet {
  public static final int MAX_BALANCE;
  private int balance;
  //@ invariant 0 <= balance & balance <= MAX_BALANCE;
  /* ... */
  //@ requires amount >= 0;
  //@ ensures balance == \old(balance) - amount & \result == balance;
  public int debit(int amount) {
    /* ... */
  }
}
```

Benefits of JML-like Annotations

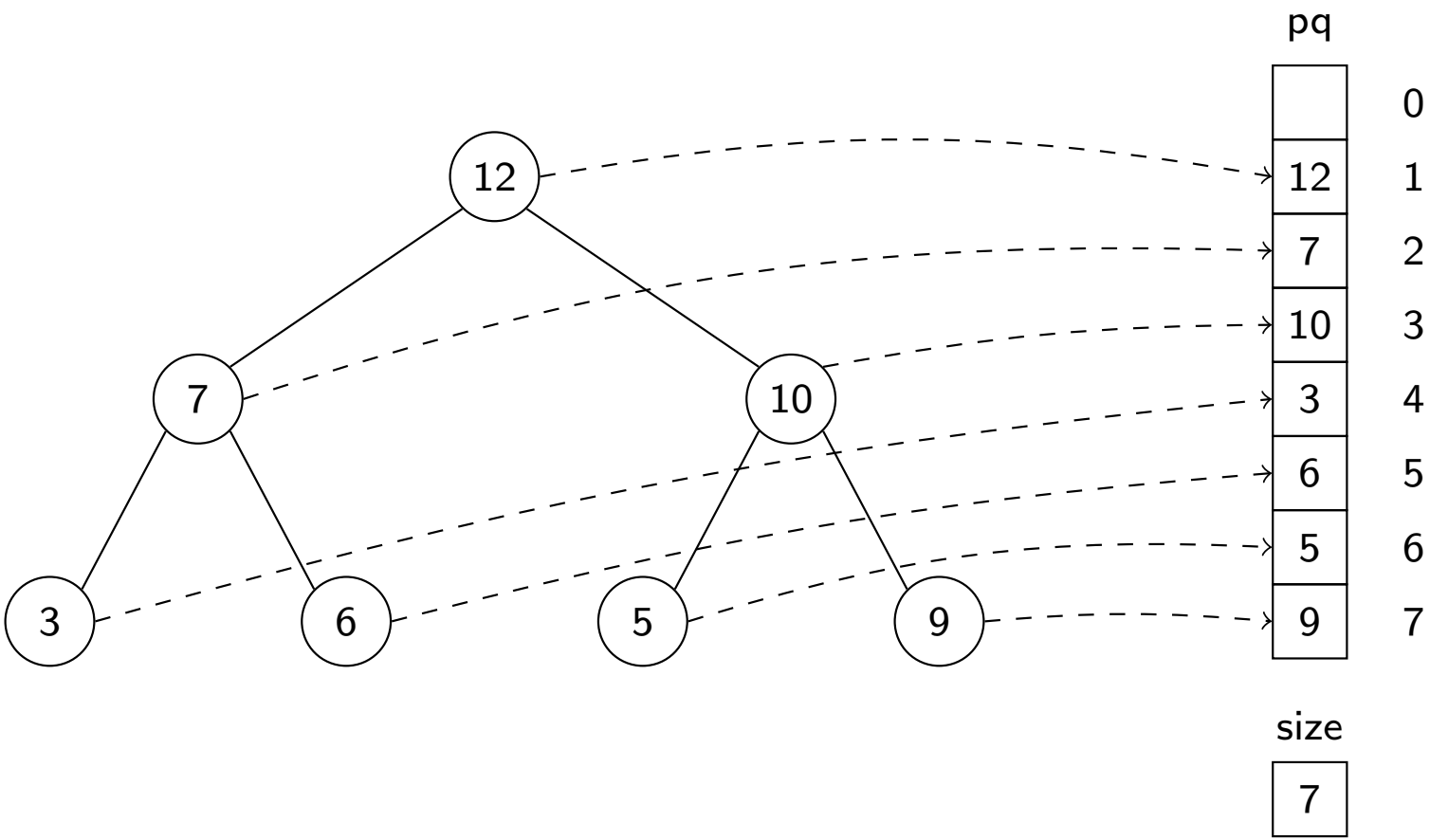
- Better documentation than simple comments
- Potentially deep specification of class and method behavior
- Via tools, annotations can be checked statically or dynamically
- Verification conditions can be extracted and proved formally
- Provides contracts to consumers of libraries

Array-based Priority Queue

```
public class PriorityQueue {
  private int[] pq;
  private int size;
  /* ... */
  public int delMax() {
    /* ... */
  }
  public void insert(int item) {
    /* ... */
  }
  public int size() {
    /* ... */
  }
  public static void main(String[] args) {
    PriorityQueue q = new PriorityQueue();
    q.insert(10);
    q.insert(17);
    q.insert(15);
    System.out.println(q.delMax()); // prints 17
  }
}
```

Array-based Priority Queue Rationale

Heap property: nodes are greater than or equal to their children



Array-based Priority Queue with Comments

```
public class PriorityQueue {
  private int[] pq; // have pq[i] >= pq[2*i] and pq[i] >= pq[2*i+1]
  private int size; // must be < pq.length


  /* ... */

  // swap out pq[1], decrease size, bubble down
  public int delMax() {
    /* ... */
  }
  // put item at pq[size + 1], increase size, bubble up
  public void insert(int item) {
    /* ... */
  }
  // just return size
  public int size() {
    /* ... */
  }
}
```

Array-based Priority Queue with JML Annotations

```
public class PriorityQueue {
  private int[] pq;
  private int size;
  //@ invariant pq != null;
  //@ invariant 0 <= size & size < pq.length;
  //@ invariant (\forall int i; 1 < i & i <= size ==> pq[i/2]>=pq[i]);
  /* ... */
  //@ requires size > 0;
  //@ ensures size == \old(size) - 1 & \result == \old(pq[1]);
  public int delMax() {
    /* ... */
  }
  //@ ensures size == \old(size) + 1 & (\exists int i; pq[i] == item);
  public void insert(int item) {
    /* ... */
  }
  //@ ensures \result == size;
  public int size() {
    /* ... */
  }
}
```

JML-style Annotation Issues

- Difficult and time-consuming to do non-trivial annotations
- Tool support[10] is lacking, in particular for Java 8
- Semantics of, e.g., Java and C is not canonically formalized

[10] http://openjml.org

Threads, Invariants, and Atomicity

- Without synchronization, invariants will not hold in presence of multiple threads
- Knowing where to use synchronization primitives is hard, but invariants can be a guide
- All public methods for a class need to be considered

Thread Safe Array-based Priority Queue using Monitors

```
public class PriorityQueue {
  private int[] pq;
  private int size;

  /* ... */

  public synchronized int delMax() {
    /* ... */
  }

  public synchronized void insert(int item) {
    /* ... */
  }

  public synchronized int size() {
    /* ... */
  }
}
```

Invariants and Atomicity

- Java synchronization is *control-centric*
  - primitives are related to methods and statements
  - programmer must consider possible thread control flows
- If an `isEmpty()` or `peekMax()` method is added, more synchronization is needed
- Easy to forget **`synchronized`** keyword and cause data races

Atomicity with Actors using Java and Akka[11]

```
public class PriorityQueue extends UntypedActor {
  private int[] pq;
  private int size;
  /* ... */
  public void onReceive(Object message) {
    // check message type, pass to method, return result as message
  }

  private int delMax() {
    /* ... */
  }

  private void insert(int item) {
    /* ... */
  }

  private int size() {
    /* ... */
  }
}
```

[11] http://doc.akka.io/docs/akka/2.0/java/untyped-actors.html

Part II: Data-centric Synchronization

Pluggable Type Systems

- **Type qualifiers** can be "plugged" into existing programs
- Pluggable type systems[12] are orthogonal to underlying types
- Examples: non-nullness, immutability, information flow, ...
- Qualifiers may be written before types, e.g., `@NonNull String`
- Included in Java 8 (JSR 308: "Type Annotations")

[12] Papi et al. (2008)

Data-centric Synchronization

- Analysing thread control flow is hard!
- A promising alternative is to focus on the *data structures*
- If the existence of an object field invariant is made explicit, synchronization can become implicit for methods
- This **data-centric synchronization** approach can be expressed as a pluggable type system
- We will use the original syntax (not JSR 308 annotations)

Timeline of Data-centric Synchronization

2006 First proposed (Vaziri et al.)

2008 Used for finding concurrency bugs (Hammer et al.)

2012 Definition of AJ dialect of Java (Dolby et al.)

2012 Static inference of AJ annotations (Huang & Milanova)

2013 Deadlock checking of AJ programs (Marino et al.)

2013 Dynamic inference of AJ annotations (Dinges et al.)

2015 Dynamic probabilistic annotation inference (Dinges et al.[13])

[13] Tech report to appear at http://osl.cs.illinois.edu

Priority Queue with Data-centric Synchronization in AJ

```
public class PriorityQueue {
  atomicset Q;
  private atomic(Q) int[] pq;
  private atomic(Q) int size;
  /* ... */
  public int delMax() {
    /* ... */
  }
  public void insert(int item) {
    /* ... */
  }
  public int size() {
    /* ... */
  }
}
```

Thread Safe Sorted List using Priority Queue

```
public class SortableList {
  atomicset L;
  private atomic(L) int[] a;
  private atomic(L) int size;
  private PriorityQueue q|Q=this.L|;

  /* ... */

  public void sort() {
    for (int i = 0; i < size; i++) {
      q.insert(a[i]);
    }
    for (int i = 0; i < size; i++) {
      int k = q.delMax();
      a[size - 1 - i] = k;
    }
  }

  public void addAllSorted(unitfor(L) SortedList other) {
    /* ... */
  }

}
```

Elements of Data-centric Synchronization

Atomic Set

Group of fields in a class connected by a consistency invariant

Example

In the `PriorityQueue` class:

- Invariant:
  - `pq` non-null,
  - `0 <= size < pq.length`, and
  - for all `i`, `1 < i <= size` implies `pq[i/2] >= pq[i]`
- Atomic set:

  `Q` = { `pq`, `size` }

Elements of Data-centric Synchronization

Atomic Set

Group of fields in a class connected by a consistency invariant

Unit of Work

Method that preserves the invariant when executed sequentially

Example

In the `PriorityQueue` class:

- Instance methods `delMax()`, `insert()`, and `size()` are units of work for the atomic set `Q`

In the `SortableList` class:

- `addAllSorted()` is a unit of work for the other list's atomic set `L`

Elements of Data-centric Synchronization

Atomic Set

Group of fields in a class connected by a consistency invariant

Unit of Work

Method that preserves the invariant when executed sequentially

Alias

Combines atomic sets

Example

- Class `SortableList` with field `q` and atomic set `L`
- Field declaration:

```
PriorityQueue q|Q=this.L|
```

- Atomic set `L` now contains `q.size` and `q.pq`

Intuitive Semantics of Atomic Sets

- Every object has a lock for each of its atomic sets
- All related atomic set locks must be held to execute methods
- An alias permanently merges atomic set locks in two objects
- `unitfor` declarations merge locks only for execution of methods

Benefits of Atomic Set Annotations

- Documents invariants without requiring formal specification
- Defers synchronization details and optimization to compiler
- Annotated programs can be checked statically for deadlocks[14]
- Normally, no additional annotations needed for new methods

[14] Marino et al. (2013)

Annotating Java Programs with Data-centric Primitives

- New programs can be structured to use atomic sets
- Legacy programs using monitors must be converted
- Advanced, fine-grained locking may not be gainful to convert
- Conversion requires understanding:
  - data invariants
  - existing synchronization

Annotating Java Programs with Data-centric Primitives

- New programs can be structured to use atomic sets
- Legacy programs using monitors must be converted
- Advanced, fine-grained locking may not be gainful to convert
- Conversion requires understanding:
  - data invariants
  - existing synchronization

Conversion Experience of Dolby et al. (2012)

- Takes several hours for rather simple programs
- 2 out of 6 programs lack synchronization of some classes
- 2 out of 6 programs accidentally introduced global locks

Part III: Inference of Atomic Sets and Applications

Static Analysis

Static analysis methods for inference of atomic sets:

- do not generally need input beyond program code
- are good at *propagating* initial annotations[15]
- can guarantee soundness
- can have difficulties in finding aliases and `unitfor` annotations

[15] Huang & Milanova (2012)

Dynamic Analysis

Dynamic methods for inference of atomic sets:

- bases analysis on *program traces*
- requires trace generation, e.g., by test suites or fuzzers
- enables inferring deeper program properties
- generally cannot alone guarantee soundness

Previous Work on Dynamic Inference (Dinges et al., 2013)

Proposed dynamic inference algorithm:

- records field accesses and data races between threads
- uses simple set membership criteria for classification
- infers atomic sets, aliases, and units of work
- evaluated qualitatively on preexisting AJ corpus

AJ Corpus of Dolby et al. (2012)

Program	**Description**	**kLoC**	**Cls.**
collections	OpenJDK collections	11.1	171
elevator	Elevator simulation	0.3	6
jcurzez1	Console window library	2.7	78
jcurzez2	Console window library	2.8	79
tsp2	Traveling salesman	0.5	6
weblech	Web site crawler	1.3	12

Evaluation Results

Atomic Sets

- Inferred annotations missing an atomic set for only two classes
  - one missing atomic set highlights faulty *collections* annotation
- Additional atomic sets are inferred for 24 classes
  - in three classes, they prevent inadvertent data races

Aliases

- Fails to infer aliases for three classes in total
  - in two classes, due to *jcurzez* data races
- New aliases added to 15 classes
  - one race condition prevented in *tsp2*

Evaluation Results

Units of Work

- One class lacks an incorrect unit of work declaration
- Additional unit of work declarations added to 13 classes

Evaluation Results

Units of Work

- One class lacks an incorrect unit of work declaration
- Additional unit of work declarations added to 13 classes

Discovered Issues with Algorithm

- highly dependent on trace exhaustiveness w.r.t. behavior
- brittleness, inference affected by small perturbations in traces
- algorithm does not scale to long executions
- algorithm does not consider distance between field accesses

Bayesian Dynamic Analysis

- Using Bayesian probabilistic methods[16], inference can be made robust against outlier observations
- New evidence is incorporated into existing knowledge in a structured way
- Rare spurious behavior is weighed against preponderance of contrary evidence and ultimately ignored

[16] Pearl (1988)

Bayes's Inversion Formula

Bayesian Inference Variables

H: "f and g are connected through an invariant" [Hypothesis]
e_k: "f, g accessed (non-)atomically with distance d_k" [evidence]

Consider a sequence of observations $e_1, \ldots, e_n$ w.r.t. f and g. Want to know *probability that H holds given* $e_1, \ldots, e_n$, i.e.,

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

Likelihood Ratios and Belief Updating

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

$$\text{updated info} = \text{info from observations} \times \text{original info}$$

$$\text{posterior odds} = \text{likelihood ratio} \times \text{prior odds}$$

$$O(H|e_1,\ldots,e_n) = L(e_1,\ldots,e_n|H) \times O(H)$$

Conditional Independence

If $e_1, \ldots, e_n$ are conditionally independent given H, we can write

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

and similarly for $\neg H$, whereby

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

Adding one more piece of evidence e_{n+1}, we get

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

Hence, if we have independence, know $O(H)$, and can compute $L(e_k|H)$, we can update odds on-the-fly when observing!

Conditional Independence

- Coarse-grained hypothesis space: $H \cup \neg H$
- With conditional independence, $e_1, \ldots, e_n$ should depend only on hypothesis, not on systematic external influence
- However, we have at least the following external factors:
  - workload
  - scheduler

Conditional Independence

- Coarse-grained hypothesis space: $H \cup \neg H$
- With conditional independence, $e_1, \ldots, e_n$ should depend only on hypothesis, not on systematic external influence
- However, we have at least the following external factors:
  - workload
  - scheduler

Mitigating Dependencies

- Working assumption: good workload and long executions minimize external influence
- Safe to include f, g in atomic set when there is no invariant...
- ...but may result in coarser-grained concurrency

Synopsis of an Algorithm for Probabilistically Inferring Atomic Sets, Aliases, and Units of Work

Assumptions about Input Programs

- Methods perform meaningful operations (convey *intent*)
- Fields that a method accesses are likely connected by invariant

Synopsis of an Algorithm for Probabilistically Inferring Atomic Sets, Aliases, and Units of Work

Assumptions about Input Programs

- Methods perform meaningful operations (convey *intent*)
- Fields that a method accesses are likely connected by invariant

Algorithm Idea

- Observe which pairs of fields a method accesses atomically and their distance in terms of basic operations
  - This is (Bayesian) *evidence* that fields are connected through an invariant
- Store current beliefs for all field pairs in *affinity matrices*

Affinity Matrix Example

	pq	size	a
pq	*	1528.3	0.015
size	1528.3	*	1
a	0.015	1	*

Analysis Supports Indirect Field Access and Access Paths

Indirect Access and Distance

- High-level semantic operations use low-level operations
- E.g., `get()` might call `getSize()` instead of accessing field `size`
- Propagate observed access to caller's scope
- Quantify directness of access as *distance*

Analysis Supports Indirect Field Access and Access Paths

Indirect Access and Distance

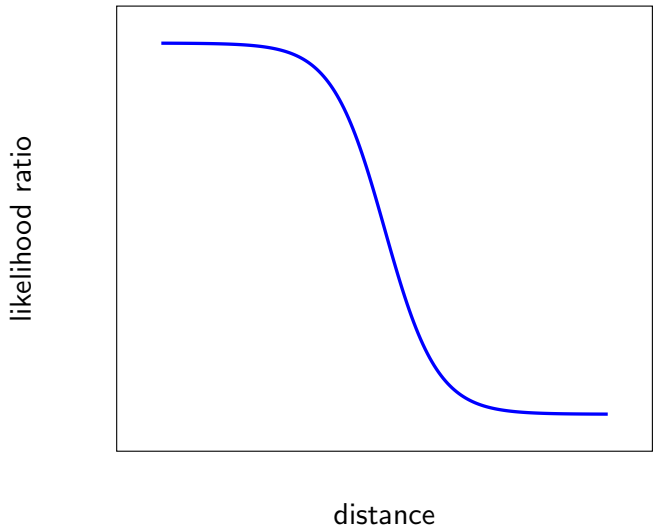
- High-level semantic operations use low-level operations
- E.g., `get()` might call `getSize()` instead of accessing field `size`
- Propagate observed access to caller's scope
- Quantify directness of access as *distance*

Access Paths

- Methods traverse the object graph
- Track *access paths* instead of field names
- Example: `this.urls.size`

Mapping Observations to Likelihoods

- Given access observation e_k for fields f and g with operation distance d_k, need to compute $L(e_k|H)$
- $L(e_k|H)$ should increase as d_k decreases up to some maximum, after which it is flat
- $L(e_k|H)$ should decrease as d_k increases down to some minimum, after which it is flat
- Result is a **logistic curve**



Algorithm in Detail: Field Access Events

Definition

A **field access event** e, captured from a program trace in the scope of a specific method call and thread, is a tuple

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

where

- Fd is the set of all fields in the program
- d is the access distance between f and g as a natural number
- $\mathrm{At} = \{\mathrm{atomic}, \mathrm{interleaved}\}$

Algorithm Details: Likelihood Ratio for Field Access Event

Definition

Let $e = (f, g, d, a)$ be a field access event. Let $\ell(d)$ be the real value defined by the logistic curve for distance d. Let p be the (real-valued, negative) data race penalty to likelihoods.

Then, the **likelihood ratio** for e is defined as

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

Algorithm in Detail: Affinity Matrices

Definition

An **affinity matrix** A is a symmetric map from pairs of fields in the program to real numbers.

$A[(f, g) \mapsto x]$ is A augmented with the mapping of (f, g) to x.

Algorithm in Detail: Affinity Matrices

Definition

An **affinity matrix** A is a symmetric map from pairs of fields in the program to real numbers.

$A[(f, g) \mapsto x]$ is A augmented with the mapping of (f, g) to x.

Example

Let $A' = A[(\mathrm{pq}, \mathrm{size}) \mapsto 1.23]$ for some affinity matrix A. Then

$$A'(\mathrm{pq}, \mathrm{size}) = A'(\mathrm{size}, \mathrm{pq}) = 1.23$$

Algorithm in Detail: Affinity Matrices

Definition

An **affinity matrix** A is a symmetric map from pairs of fields in the program to real numbers.

$A[(f, g) \mapsto x]$ is A augmented with the mapping of (f, g) to x.

Example

Let $A' = A[(\mathrm{pq}, \mathrm{size}) \mapsto 1.23]$ for some affinity matrix A. Then

$$A'(\mathrm{pq}, \mathrm{size}) = A'(\mathrm{size}, \mathrm{pq}) = 1.23$$

Definition

In an **initial affinity matrix** A^{init}, it holds that $A^{\mathrm{init}}(f, g) = 1$, for all $(f, g) \in \mathrm{Fd} \times \mathrm{Fd}$.

Algorithm in Detail: Belief Configurations

Definition

A **belief configuration** $B \in \mathrm{Config}$ is a map from methods to affinity matrices, such that B contains a matrix A_m for each method m in the program.

$B[m \mapsto A']$ is B augmented with the mapping of m to A'.

Algorithm in Detail: Belief Configurations

Definition

A **belief configuration** $B \in \mathrm{Config}$ is a map from methods to affinity matrices, such that B contains a matrix A_m for each method m in the program.

$B[m \mapsto A']$ is B augmented with the mapping of m to A'.

Definition

In an **initial belief configuration** B^{init}, it holds that, for all methods m, $B^{\mathrm{init}}(m) = A^{\mathrm{init}}$.

Algorithm in Detail: Configuration Transition Function

Definition

Let $e = (f, g, d, a)$ be a field access event for a thread t and method m. 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 defined as

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

where

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

Algorithm in Detail: Final Belief Configuration

Definition

Let $e_1, \ldots, e_n$ be all field access events in a trace for the thread t and method m. Then, the algorithm's **final belief configuration** for t and m is defined as

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

Algorithm Properties

- Likelihoods incorporate scope and distance of observations
- Beliefs are revised by new evidence, i.e., can improve over time
- Analysis becomes robust and insensitive to outlier observations
- Observation data in codebase size, not trace size
- Complements static analysis with `unitfor` and aliases

Inference Example

```
public class List {
  private int size;
  private Object[] elements;

  public int size() {
    return size;
  }

  public Object get(int i) {
    if (0 <= i && i < size)
      return elements[i];
    else
      return null;
  }
  /* ... */
}
```

```
public class DownloadManager {
  private 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;
  }
  /* ... */
}
```

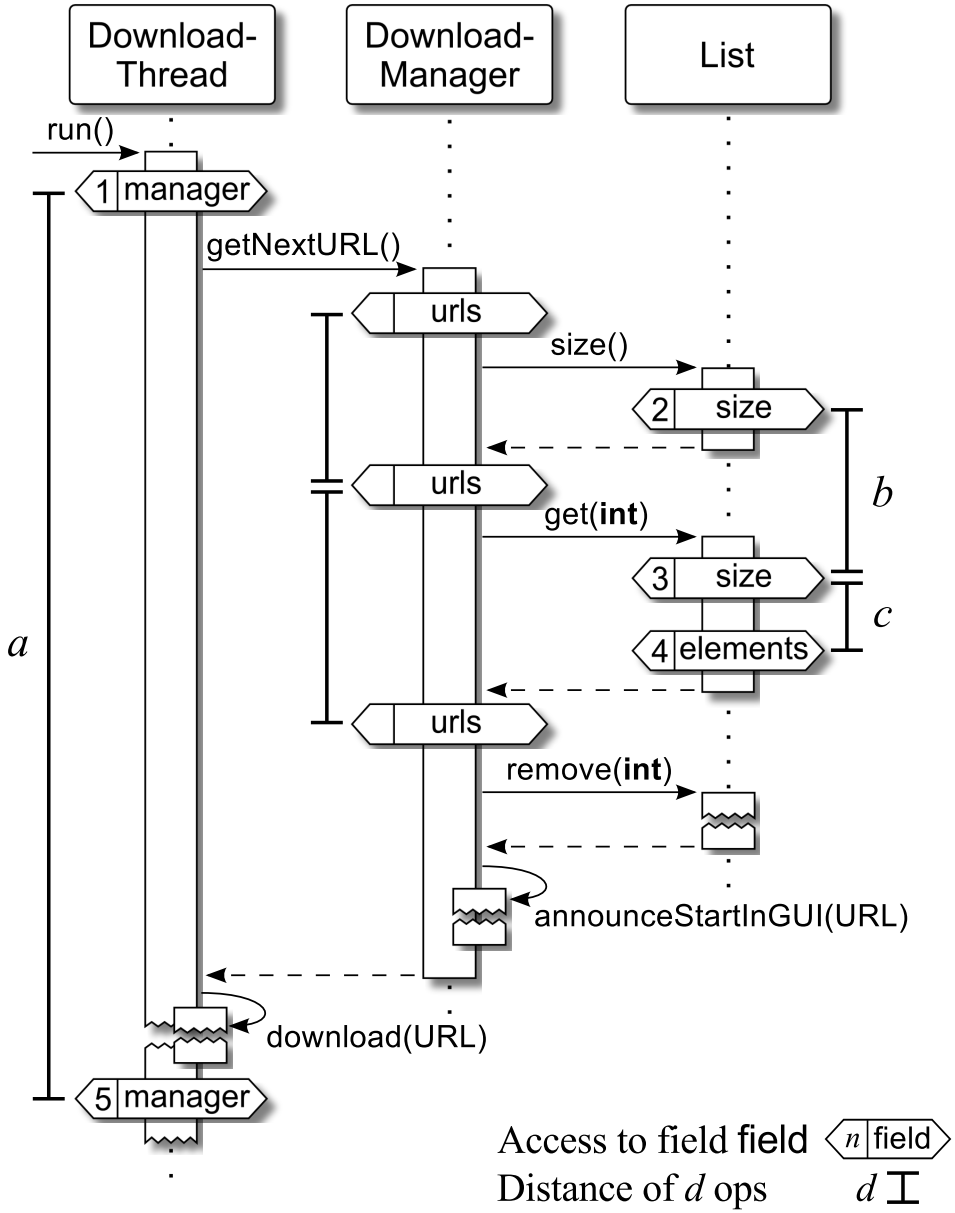
Inference Example

```
public class DownloadThread extends Thread {
  private DownloadManager manager;

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

Inference Example Driver Code

```
public class Download {
  public static void main(String[] args) {
    DownloadManager manager = new DownloadManager();
    for (int i = 0; i < 128; i++) {
      manager.addURL(new URL("http://www.example.com/f" + i));
    }
    DownloadThread t1 = new DownloadThread(manager);
    DownloadThread t2 = new DownloadThread(manager);
    DownloadThread t3 = new DownloadThread(manager);
    t1.start();
    t2.start();
    t3.start();
  }
}
```


Download-Thread
Download-Manager
List
run()
1 manager
getNextURL()
urls
size()
2 size
urls
get(int)
3 size
4 elements
urls
remove(int)
announceStartInGUI(URL)
download(URL)
5 manager
a
b
c
Access to field field
n field
Distance of d ops
d

Inference Example Results

```
public class List {
  atomicset L;
  private atomic(L) int size;
  private atomic(L) Object[]
      elements;

  public int size() {
    return size;
  }

  public Object get(int i) {
    if (0 <= i && i < size)
      return elements[i];
    else
      return null;
  }
  /* ... */
}
```

```
public class DownloadManager {
  atomicset U;
  private List urls|L=this.U|;

  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;
  }
  /* ... */
}
```

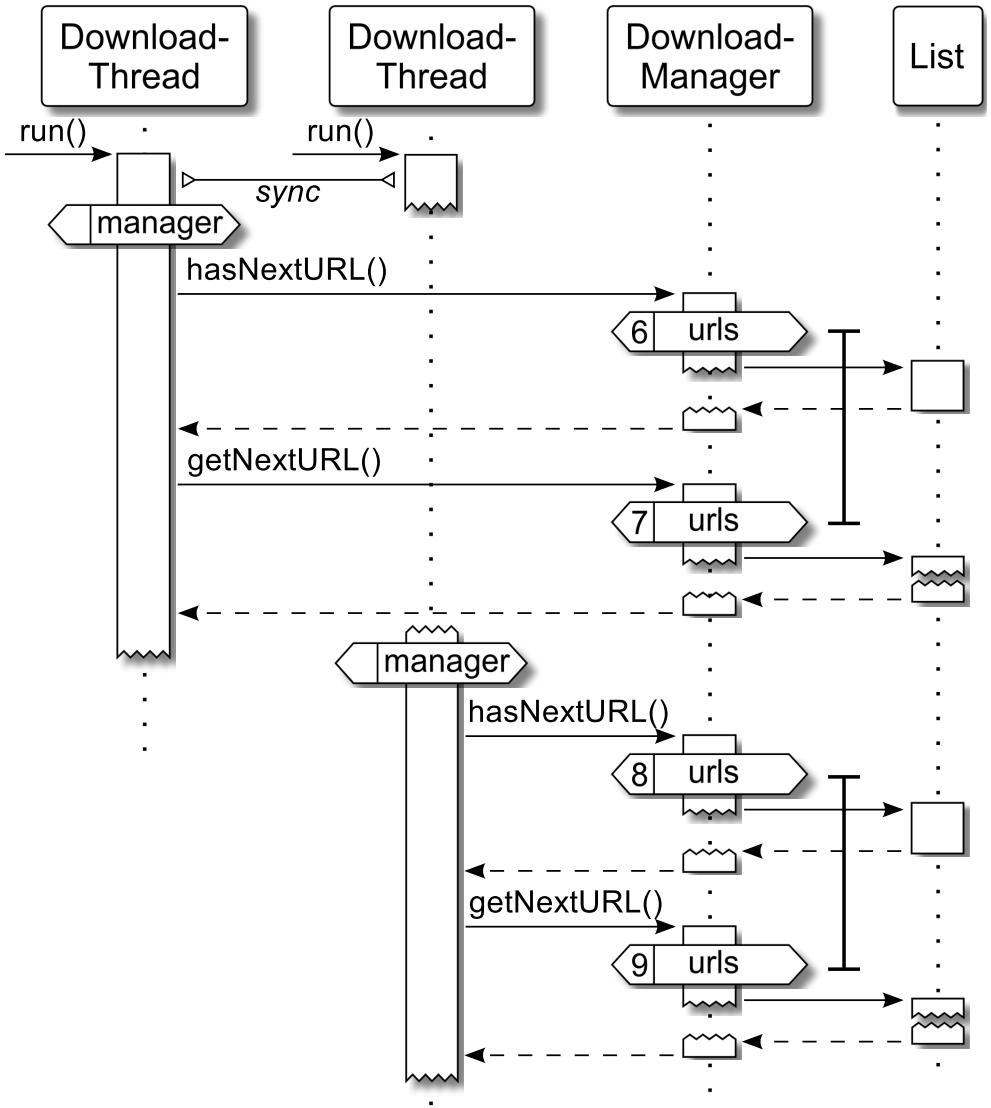
Inference Example After Removal of Monitors

```
public class DownloadThread extends Thread {
  private DownloadManager manager;

  public void run() {
    while (true) {
      URL url;
      if (!manager.hasNextURL())
        break;
      url = manager.getNextURL();
      download(url); // Blocks while waiting for data
    }
  }
  /* ... */
}
```

Inferring Aliases Can Unduly Constrain Concurrency

- Aliases can introduce global locks!
- Assume an atomic set `T` is added to `DownloadThread`
- Assume `T` is aliased to `U` in `DownloadManager`
- `run()` is then unit of work for atomic set spanning threads
- Heuristic is used to avoid inferring such aliases


Download-Thread
Download-Thread
Download-Manager
List
run()
run()
sync
manager
hasNextURL()
6 urls
getNextURL()
7 urls
manager
hasNextURL()
8 urls
getNextURL()
9 urls

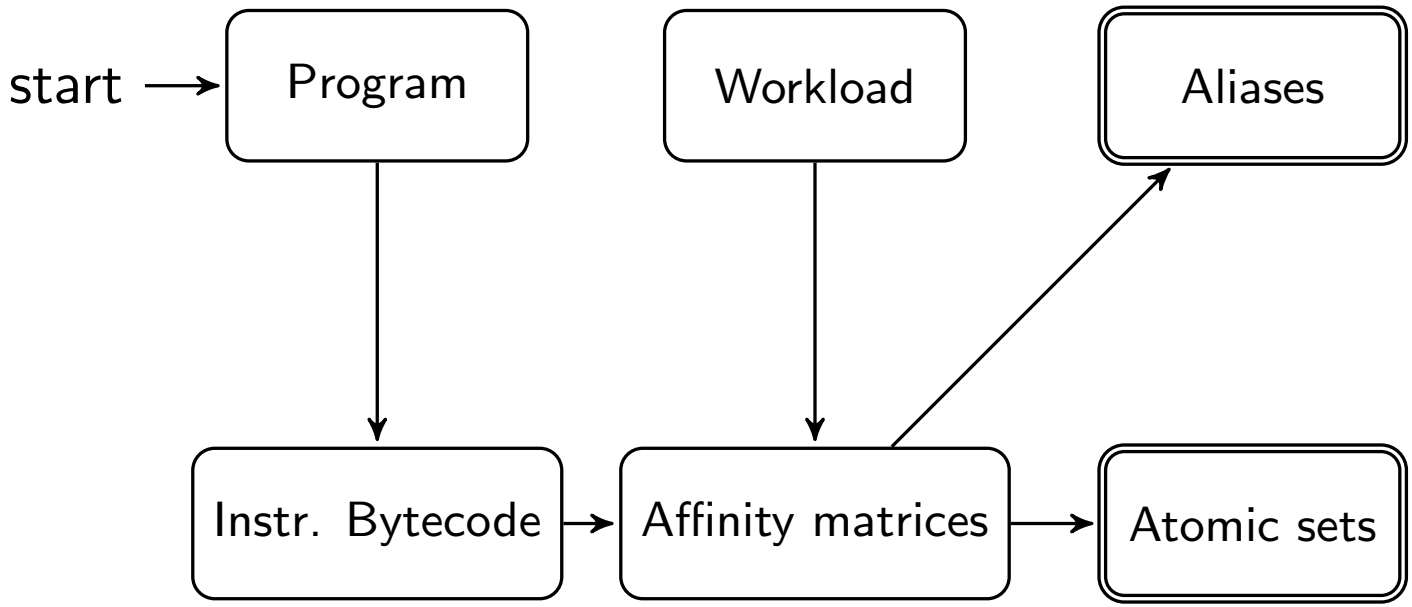
Implementation

Our proof-of-concept algorithm implementation consists of:

- a byte code instrumenter using WALA's[17] Shrike library
- an inferencer

[17] http://wala.sf.net

Data-Centric Synchronization Implementation Toolchain



Algorithm Parameter Tuning

Many tool parameters must be tuned in practice:

- penalty for data races
- distance for language constructs
- logistic function exponent
- odds cutoff point for adding atomic sets

Evaluation Results

- Inference scales in codebase size; *collections* inference finishes in less than one minute on Intel Core i5 laptop
- Overall, inferred annotations mostly agree with the manual annotations from corpus, and add more annotations that document behavior

Evaluation Results

collections Tool infers almost all atomic sets. Some fields are missing in sets due to workload issues; one omitted field avoids a global lock. Almost all units of work inferred, with missing units mostly due to fuzzer.

elevator Tool does not infer manual units of work, but these annotations do not conform to AJ specification. One alias is missing. Several atomic sets and units of work added that document behavior.

tsp2 Tool infers all manual annotations, while adding atomic sets and aliases that document existing behavior.

Evaluation Results

jcurzez1 Tool correctly infers all manually added atomic sets for all but one class, where one set is missing a field due to a conservative choice of alias inference parameters. One new atomic set and one unit of work are added that prevent inadvertent races. Some manual units of work annotations are not inferred due to incomplete workloads.

jcurzez2 Tool infers all manually added atomic sets for all but one class, as for *jcurzez1*, while adding atomic sets and aliases that document behavior. Some manual units of work annotations are not inferred due to incomplete workloads. Two added units of work prevent inadvertent races.

weblech Tool infers all manual annotations, while adding atomic sets and aliases that document behavior.

Actorizing Programs Annotated with Atomic Sets

- Key property: messages to actors are processed *one at a time*
- Fields in one atomic set *should not* span two actors at runtime
- An actor encapsulates one or more objects with atomic sets
- Actor-wrapped objects must be accessed through interfaces

Actorizing Programs Annotated with Atomic Sets

- Key property: messages to actors are processed *one at a time*
- Fields in one atomic set *should not* span two actors at runtime
- An actor encapsulates one or more objects with atomic sets
- Actor-wrapped objects must be accessed through interfaces

Akka's Typed Actors for Java

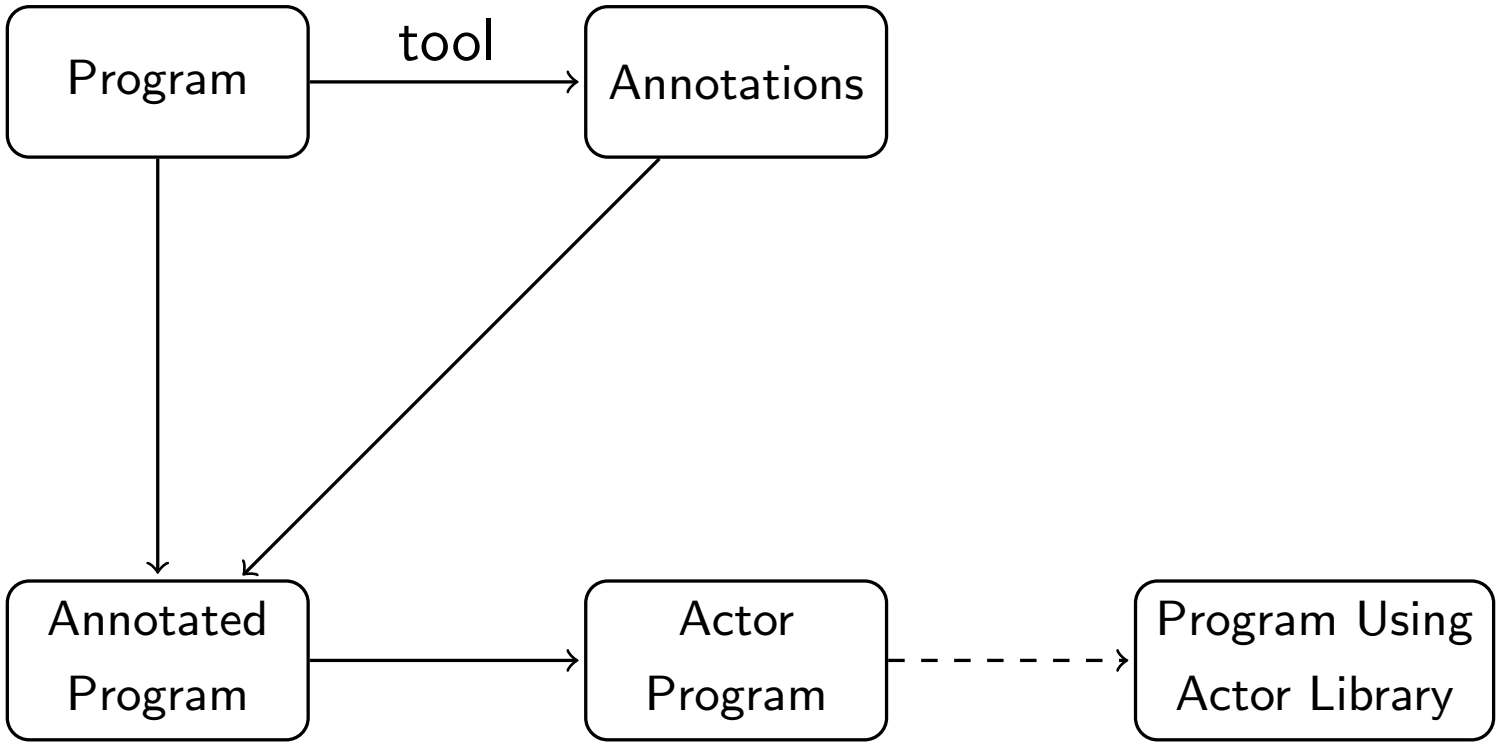
- Akka allows mixing objects and actors through Typed Actors[18]
- Method calls to actors automatically translates to messages
- Arguments in messages should be actor refs or immutable!

[18] http://doc.akka.io/docs/akka/2.0/java/typed-actors.html

Actorizing Programs Annotated with Atomic Sets

- When object instances are created:
  - instances of class `Thread` end up in separate actor
  - instances with *non-aliased* atomic sets end up in separate actor
  - instances with *aliased* atomic sets end up inside same actor
- Explicit synchronization (e.g., two-phase commit) needed to handle some `unitfor` declarations!

Conversion Approach in Practice



Conversion Example

```
public class List {
  atomicset L;
  private atomic(L) int size;
  private atomic(L) Object[]
      elements;

  public int size() {
    return size;
  }

  public Object get(int i) {
    if (0 <= i && i < size)
      return elements[i];
    else
      return null;
  }
  /* ... */
}
```

```
public class DownloadManager {
  atomicset U;
  private List urls|L=this.U|;

  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;
  }
  /* ... */
}
```

```
public class DownloadThread extends Thread {
  private DownloadManager manager;
  public void run() {
    while (true) {
      URL url;
      if (!manager.hasNextURL()) break;
      url = manager.getNextURL();
      download(url);
    }
  }
  /* ... */
}
public class Download {
  public static void main(String[] args) {
    DownloadManager manager = new DownloadManager();
    for (int i = 0; i < 128; 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();
   }
}
```

Conversion Rationale

- `DownloadManager` instance has atomic set, becomes actor in `main()` – interface must be extracted
- `List` instance is aliased, does not become actor in `DownloadManager`
- `DownloadThread` is a subclass of `Thread`, instances become actors in `main()` – interface must be extracted

Conversion Results

```
public class List {
  private int size;
  private Object[] elements;

  public int size() {
    return size;
  }

  public Object get(int i) {
    if (0 <= i && i < size)
      return elements[i];
    else
      return null;
  }
  /* ... */
}
```

```
public class DownloadManager
    implements IDownloadManager {
  private 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;
  }
  /* ... */
}
```

```
public class DownloadThread extends Thread implements IDownloadThread {
  private IDownloadManager manager;
  public void run() {
    while (true) {
      URL url;
      if (!manager.hasNextURL()) break;
      url = manager.getNextURL();
      download(url);
    }
  }
  /* ... */
}
public class Download {
  public static void main(String[] args) {
    IDownloadManager manager = new actor DownloadManager();
    for (int i = 0; i < 128; i++) {
      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();
   }
}
```

```
public class Download {
  public static void main(String[] args) {
    final IDownloadManager manager =
       create(IDownloadManager.class,
        new Creator<IDownloadManager>() {
          @Override
          public IDownloadManager create() throws Exception {
            return new DownloadManager();
          }
        });
    for (int i = 0; i < 128; i++) {
      manager.addURL(new URL("http://www.example.com/f" + i));
    }
    IDownloadThread t1 =
       create(IDownloadThread.class,
        new Creator<IDownloadThread>() {
          @Override
          public IDownloadThread create() throws Exception {
            return new DownloadThread(manager);
          }
        });
    /* ... */
   }
}
```

Atomic Sets for Objects vs. Synchronizers for Actors

Atomic sets

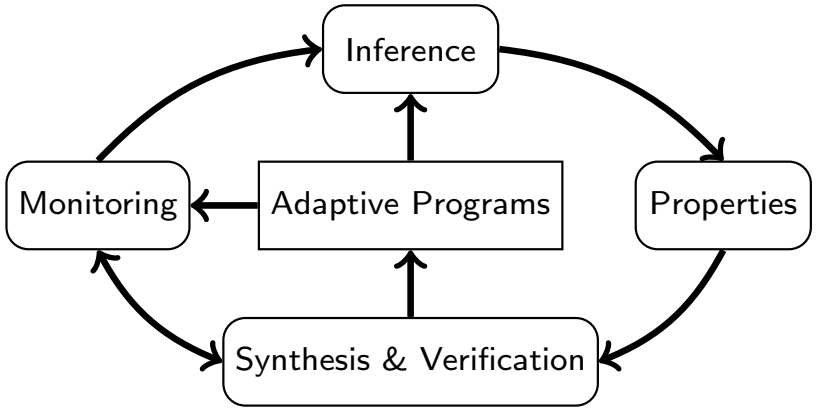
- provide *spatial* atomicity
- support transitive extensions of atomicity via aliases
- prevents interleaved access to shared data

Synchronizers

- can provide *temporal* atomicity (messages arrive together)
- do not directly support transitive extension similar to aliases
- cannot easily express non-interleaving of message sequences

Ongoing and Planned Future Work

- Public software release of probabilistic inference tool
- Actor implementation of `unitfor` using two-phase commits
- Code refactorings for atomic sets and better actorization
- Tuning and evaluation of actorization via atomic set inference
- Integration of monitoring, static & dynamic inference, and synthesis into a feedback loop for *program adaptation*



Needed: Robust, Standalone AJ-to-locks Compiler

- Most straightforwardly, a Java source-to-source translation
- Old compiler relied on Eclipse, new should be standalone
- Several atomic sets can be implemented by single lock
- Initial version suitable as master's level project

Acknowledgements

This research has been funded in part by:

NSF grant number CCF-1438982

AFOSR contract FA 9750-11-2-0084

Bibliography I

Cyrille Artho, Klaus Havelund, and Armin Biere.
High-level data races.
In *NDDL '03*, pages 82–93. ICEIS Press, 2003.

Peter Dinges, Minas Charalambides, and Gul Agha.
Automated inference of atomic sets for safe concurrent execution.
In *PASTE '13*, pages 1–8, 2013.

Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek.
A data-centric approach to synchronization.
ACM TOPLAS, 34(1):4, 2012.

Svend Frølund and Gul Agha.
A language framework for multi-object coordination.
In *ECOOP '93*, pages 346–360. Springer, 1993.

Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip.
Dynamic detection of atomic-set-serializability violations.
In *ICSE '08*, pages 231–240, 2008.

Bibliography II

C. A. R. Hoare.
Monitors: An operating system structuring concept.
Commun. ACM, 17(10):549–557, 1974.

Wei Huang and Ana Milanova.
Inferring AJ types for concurrent libraries.
In *FOOL '12*, pages 82–88, 2012.

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou.
Learning from mistakes: a comprehensive study on real world concurrency bug characteristics.
In *ASPLOS '08*, pages 329–339. ACM, 2008.

Daniel Marino, Christian Hammer, Julian Dolby, Mandana Vaziri, Frank Tip, and Jan Vitek.
Detecting deadlock in programs with data-centric synchronization.
In *ICSE '13*, pages 322–331, 2013.

Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst.
Practical pluggable types for Java.
In *ISSTA '08*, pages 201–212, 2008.

Bibliography III


Judea Pearl.
Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.
Morgan Kaufmann Publishers Inc., 1988.


Weslley Torres, Gustavo Pinto, Benito Fernandes, João Paulo Oliveira, Filipe Alencar Ximenes, and Fernando Castor.
Are Java programmers transitioning to multicore?: A large scale study of Java FLOSS.
In *SPLASH '11 Workshops*, pages 123–128, 2011.


Mandana Vaziri, Frank Tip, and Julian Dolby.
Associating synchronization constraints with data in an object-oriented language.
In *POPL '06*, pages 334–345, 2006.