

Assuring and Evolving Concurrent Programs: Annotations and Policy

Aaron Greenhouse
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
aarong@cs.cmu.edu

William L. Scherlis
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
wls@cs.cmu.edu

ABSTRACT

Assuring and evolving concurrent programs requires understanding the concurrency-related design decisions used in their implementation. In Java-style shared-memory programs, these decisions include which state is shared, how access to it is regulated, the roles of threads, and the policy that distinguishes desired concurrency from race conditions. These decisions rarely have purely local manifestations in code.

In this paper, we use case studies from production Java code to explore the costs and benefits of a new annotation-based approach for expressing design intent. Our intent is both to assist in establishing “thread safety” attributes in code and to support tools that safely restructure code—for example, shifting critical section boundaries or splitting locks. The annotations we use express “mechanical” properties such as lock-state associations, uniqueness of references, and encapsulation of state into named aggregations. Our analyses revealed race conditions in our case study samples, drawn from open-source projects and library code.

The novel technical features of this approach include (1) flexible encapsulation via aggregations of state that can cross object boundaries, (2) the association of locks with state aggregations, (3) policy descriptions for allowable method interleavings, and (4) the incremental process for inserting, validating, and exploiting annotations.

1. INTRODUCTION

A programmer trying to understand and safely evolve a Java-style concurrent program needs to answer a number of questions of non-local character. For example: *What portions of state are shared and which locks protect them? Who has responsibility, caller or callee, to acquire a particular lock? Which interleavings can occur and which of these are acceptable from the standpoint of data integrity?* The non-local character of these questions means that their answers

cannot in general be readily gleaned from code [33, 24].

Without answers to these questions, it is difficult for programmers to make even simple evolutionary changes to concurrent code. Programmers may be forced to make guesses concerning design policy. For example, which lock should be used to protect a shared resource, and what portions of state comprise the resource being shared? Also, are all accesses to that shared state protected by critical sections? The inability to develop and appropriately document the answers to these questions can complicate the evolution of shared-memory concurrent programs in languages such as Java and Ada 95. Creating even a simple Java applet entails coordinating the actions of multiple threads according to informally stated policies of the published API.

Contributing to this problem are the lack of practical intermediate notations and methods for expressing and reasoning about routine concurrency properties. Notations and diagrams exist and are widely used for other aspects of design knowledge, including architecture [19], class and object design, sequencing, and state transition [6].

In this paper, we consider an approach to expressing design intent that enables programmers to obtain precise answers, and to do so in a way that enables assured consistency between expressed design intent and code. This approach is based on three key elements: (1) formal program annotations concerning “mechanical” program properties, significantly extending ideas from [10, 14, 13]; (2) maintainable consistency between implementation and design intent via composable static program analyses, building on ideas from [22] and other sources; and (3) a step-by-step process for recording design intent and establishing assurance. In particular, increments of assurance are obtained from increments of effort in declaring annotations and policy, and in carrying out the analyses that relate annotations and policy with source code.

Our approach is based on a key hypothesis: safe evolution of concurrent production code can be carried out with less explicit up-front effort using this kind of incremental approach than in approaches that rely on full functional specification. Concurrent properties are of particular significance in software quality assurance because they can be difficult to assure using traditional testing approaches [4, 25].

The novel technical ideas presented in this paper include (1) flexible encapsulation via aggregations of state that can cross object boundaries, (2) the explicit association of locks with these state aggregations, (3) concise policy descriptions

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

ICSE'02, May 19-25, 2002, Orlando, Florida, USA.

Copyright 2002 ACM 1-58113-472-X/02/0005...\$5.00.

for allowable method interleavings, and (4) an incremental process for inserting, validating, and exploiting annotations. The principal purpose of the paper, however, is to combine these ideas into a practicable approach for documenting concurrency-related design intent and providing ongoing assurance as programs evolve.

We recognize that there is an “expression cost” associated with any approach that requires programmers to make design knowledge explicit in notations or diagrams. We limit the required expression to the kinds of mechanical properties generally familiar to programmers (as evident from informal documentation in code, for example), and use a simple syntax. The other design notations mentioned above have been adopted widely enough to suggest this goal is realistic.

1.1 Annotations, Policy, and Thread Safety

In Section 3, we describe annotations that express several aspects of low-level design intent. The annotations focus on concurrency-related aspects of the contract of a component with its clients. Composable versions of familiar analyses enable ongoing reconciliation of the expressed intent with implemented code. Most of these aspects are variants of well known concepts in program analysis. Specifically, annotations

- Name and hierarchically organize the state of a program, with aggregates that may span multiple objects.
- Describe which state is affected by a method (or other code segment), and what is the nature of the effects.
- Describe uniqueness of object references.
- Associate locks with abstract aggregations of state, and provide names for the locks.
- Specify which methods may be executed concurrently.
- Delineate responsibility for acquiring locks (e.g., caller vs. callee).

In our examples, we demonstrate how these annotations can support the establishment of an intuitive notion of thread safety. Of course, thread safety is not a precise concept: it is generally taken to mean that clients may only access a resource when the representation invariants of the resource are intact. This definition is problematic because representation invariants are rarely expressed (and even more rarely formalized) in production software.

Our approach is designed to accommodate this reality by offering programmers a concise and operationally oriented way to express two key design-related consequences of reasoning with invariants. These are conventions concerning lock–state relationships and conventions concerning safe interleaving policy.

Policy conventions enable, for example, relating the management of a lower-level resource (e.g., `java.lang.Vector`) with design goals for a higher-level resource. For example, the `length`, `get`, and `remove` methods of `Vector` do not interfere with each other because the implementation of `Vector` uses locks. But two threads sharing a `Vector` must coordinate at a higher level—one thread removing an element can interfere with another thread iterating over a vector. In other words, safe implementation of a shared low-level resource does not imply correct use of that resource with respect to the invariants of a higher-level client resource.

In our approach, the low-level design knowledge is expressed in the form of annotations. There is evidence of programmer adoption of tool-supported mechanical annota-

tion: consider Microsoft’s Hungarian notation [36], Eiffel’s design-by-contract [31], Java’s `final` class and method modifier, and exception declarations in method signatures.

Although our approach is not as expressive as assertions in a logic of programs (cf. ESC/Java [28]), it operationalizes several important aspects. In particular, the validity of most annotations can be established using composable program analysis techniques. Annotations serve as cut points, enabling global analysis to be avoided and opaque components to be integrated safely, contingent on the validity of their annotations. Composable analyses link annotations together, enabling step-by-step construction of “chains of evidence” of the validity of hypotheses embodied in annotations. Suitable analyses have been formulated, for example, for object-oriented effects and unique references [22, 8]. In addition, the use of annotations can give performance benefits—an overly defensive approach to locking (e.g., synchronizing all constructors and instance methods of a class) can be replaced by a precise placement of critical sections in which there is less lock acquisition at runtime.

1.2 This Paper

We introduce an ongoing example in Section 2, in which we demonstrate our techniques by applying them to code from Apache Log4j. The annotation language is described in detail in Section 3. There is considerable related work, which we discuss in the next section. We then consider prospects for the integration of these ideas into software engineering practice, with emphasis on (1) step-by-step creation of chains of evidence of “safe” concurrency, and (2) ongoing assurance during program evolution.

Our notation and examples are based in Java. Java’s approach to shared-memory lock-based concurrency is typical among modern languages. Concurrent access to data is coordinated by explicitly acquiring and implicitly releasing locks using `synchronized` blocks. Any object may be used as a lock. Java provides no syntactic mechanism for distinguishing shared data from non-shared data, and there is no notation for associating data with the lock that is supposed to protect access to it. Every object may also be used as a signal-and-continue condition variable via the methods `wait`, `notify`, and `notifyAll`.

2. AN EXAMPLE: BOUNDED FIFO

We introduce as an ongoing example the class `BoundedFIFO` from the Log4j debug/logging library [2]. The use of annotations will make explicit: (1) the delineation of the shared state of an abstract `BoundedFIFO`, (2) the locking conventions for safely accessing the state, (3) the placement of responsibility to acquire locks, and (4) which methods are safe to execute concurrently.

In the library, `LoggingEvents` are passed from event sources to event listeners. Listeners use `BoundedFIFO` instances to buffer received events—see Figure 1¹; an annotated version is in Figure 5. `BoundedFIFO` implements a circular queue using an array `buf`, with head and tail indices `first` and `next`. The buffer is non-blocking: `getting` from an empty buffer returns `null`, and `putting` to a full buffer silently drops the new event. Client-side blocking is facilitated by the methods `isFull`, `wasFull`, and `wasEmpty`.

¹Log4j is Copyright ©1999 The Apache Software Foundation.

```

public class BoundedFIFO {
    LoggingEvent[] buf;
    int numElts = 0, first = 0, next = 0, size;

    /** Create a new buffer of the given capacity. */
    public BoundedFIFO(int size) {
        if(size < 1) throw new IllegalArgumentException();
        this.size = size;
        buf = new LoggingEvent[size];
    }

    /** Returns <code>null</code> if empty. */
    public LoggingEvent get() {
        if(numElts == 0) return null;
        LoggingEvent r = buf[first];
        if(++first == size) first = 0;
        numElts--;
        return r;
    }

    /** If full, then the event is silently dropped. */
    public void put(LoggingEvent o) {
        if(numElts != size) {
            buf[next] = o;
            if(++next == size) next = 0;
            numElts++;
        }
    }

    /** Get the capacity of the buffer. */
    public int getMaxSize() { return size; }

    /** Get the number of elements in the buffer. */
    public int length() { return numElts; }

    /** Returns <code>true</code> if the buffer was empty
     * before last put operation. */
    public boolean wasEmpty() {
        return numElts == 1;
    }

    /** Returns <code>true</code> if the buffer was full
     * before the last get operation. */
    public boolean wasFull() { return numElts+1 == size; }

    /** Is the buffer full? */
    public boolean isFull() { return numElts == size; }
}

```

Figure 1: Unannotated class BoundedFIFO.

Clients must follow an *undocumented* convention during concurrent use to assure atomic access to the FIFO. For example, two concurrent executions of `put` could result in the loss of an event, because the two different events could be written to the same `buf` location. The synchronization convention is that access to a `BoundedFIFO` instance must be coordinated by clients synchronizing on the instance. Typical usage requires a critical section within the client spanning a series of calls to a `BoundedFIFO`.

In the next section we describe how annotations can be added to ensure that the class is used appropriately. Figure 2 reorganizes *client* code from `Log4j` into a single canonical, correct client for `BoundedFIFO`.

3. ANNOTATIONS

A fundamental requirement for assuring the correct use of shared state is the identification of state that is shared. This state may encompass multiple variables and span multiple objects. The aggregations that we name are meant to correspond with abstract state in low-level design, even though their representation in code may be spread across parts of multiple entities.

Once we can give names to state, we can consider the association of locks with state. We must also identify conventions for who does the locking. Finally, we describe policy specifications for expressing design intent concerning allowable interleavings. We build on previous results related to effects and uniqueness annotations and analyses, which are briefly summarized in this section in order to permit our overall presentation to be self-contained.

```

public class FIFOClient { ...
    private final BoundedFIFO fifo = ...;
    ...
    public void putter(LoggingEvent e) {
        synchronized(fifo) {
            while(fifo.isFull()) {
                try { fifo.wait(); }
                catch(InterruptedException ie) { }
            }
            fifo.put(e);
            if(fifo.wasEmpty()) fifo.notify();
        }
    }

    public LoggingEvent getter() {
        synchronized(fifo) {
            LoggingEvent e;
            while(fifo.length() == 0) {
                try { fifo.wait(); }
                catch(InterruptedException ie) { }
            }
            e = fifo.get();
            if(fifo.wasFull()) fifo.notify();
            return e;
        }
    }

    public int length() {
        synchronized(fifo) { return fifo.length(); }
    }
}

```

Figure 2: Example client of BoundedFIFO.

3.1 Regions and State Aggregations

The programmer must be able to name sections of program state in order to mark it as shared. This is challenging because (1) not all state is explicitly named by program variables, and (2) inappropriate use of private or local program names can violate principles of encapsulation and frustrate program evolution. The approach in this paper to naming state extends the object-oriented effects system described in [22] to include parameterization.

3.1.1 Region annotations

To help answer *what is the program state*, the programmer defines names for extensible groups of fields (Java instance and class variables) called *regions*. Publicly visible regions represent the abstract data manipulated by the abstract operations of the class. For example, a 2-D Point class might place its `x` and `y` fields within a `Location` region; a `Point3D` subclass could extend that region with a new `z` field, and a `ColorPoint` subclass could locate a new `color` field within a new `Appearance` region. Regions can be identified incrementally: only those classes relevant for a particular concern need to be annotated with region naming, and that naming structure can be refined without disturbing existing conclusions. Defaulting reduces the extent of required annotation.

The intent is that programmers are able to define regions so that fields associated with the same abstract design concept are identified with the same region. This produces a region *hierarchy*, which, roughly, is an abstract overlay on the class hierarchy, though the aggregations do not respect the class hierarchy. This overall structure is key to the scalability of our approach.

The visibility and the class- or instance-level nature of a region may be specified using the standard modifiers. Java field definitions define the leaves of the region hierarchy. Several regions are predefined, including `Instance`, the default parent of instance regions for a class; `Static`, the default parent of `static` (class-level) regions; and `[]` (pronounced *element*), a subregion of `Instance` that contains array elements. Regions are inherited by subclasses (subject to visibility constraints) which may extend the membership of already named inherited regions. Regions are declared

using the annotation

```
/*@ visibility region region in parent */
```

where the clause “in parent” is optional.² Fields are assigned to a non-default parent region by appending

```
/*@ in parent */
```

to their declaration.

Class `BoundedFIFO` in Figure 1 is simple enough that the default region `Instance` suffices for naming the set of fields that comprise the abstract shared state of a `BoundedFIFO`. In Section 3.1.3, a different example is presented that demonstrates user-defined regions.

In Java, the handle on an object is a reference. *Targets* are an extrinsic mechanism to name *references* to regions. If variable `p2d` refers to a 2-D Point, then instance target `p2d.Location` identifies the `x` and `y` fields of *that object*. But if `p2d` refers to a `Point3D`, then that same target identifies the `x`, `y`, and `z` fields. In annotations, targets are used only in effects declarations and as actual parameters in region polymorphism.

In analysis, two targets may be compared to determine if they have a non-null intersection of state. This comparison of targets involves alias analysis among program variables because multiple targets may refer to the same region: the region identified by target `p2d.Location` is distinct from the region identified by target `p2d.2.Location` only when `p2d` and `p2d.2` are unaliased. Similarly, aliasing can cause a single target to resolve to multiple possible regions. (This ambiguity motivates our use of unique annotations—see below—to enable alias analyses to be less conservative [8].) The results of target comparison in the absence of subclasses are not violated by the introduction of subclasses. This is because we require subclass definitions to respect effects declarations in superclasses (see below).

3.1.2 Aggregation through uniqueness

The state of a single design abstraction often spans the state of multiple instantiated objects. We have two techniques for aggregating state from many objects into a single region. The first, introduced in [22], is based on the exploitation of unaliased (“unique”) references. We summarize this technique here for completeness, though this is not a new result. (The second technique, presented in the next section, is a new approach to aggregation based on parameterizing classes by regions to enable reasoning about abstract structures spanning multiple objects.)

Normally, the `Instance` region does not include the state of referenced objects. Returning to our example, our intent for `BoundedFIFO` is that the shared state also include the entire separate array object referenced by the field `buf`. The array is problematic because it might have aliases, and protecting the *reference* with a lock does not guard against simultaneous access to the array object. If the array is not protected, we cannot assure integrity of the FIFO elements. A conservative effects analysis is used to provide assurance of locking conventions. If it cannot prove that the array is unaliased then it would report *all* arrays as affected by `put` and `get`. Assuring safe concurrency in this case would require all arrays in the entire system using a `BoundedFIFO` to be protected by a single lock.

The array referenced by `buf` is, of course, not intended to be shared: the reference is meant to be *unique*. We record

²Following Java convention, annotations are embedded in comments beginning with the at-sign “@.”

this fact by inserting annotations into the implementation of `BoundedFIFO` so the effects system can exploit it:

```
/*@unique*/ LoggingEvent[] buf; /*@ {} in Instance
```

There are two annotations here: (1) the field is unique, and (2) the elements of the array object, residing in the `[]` region, are declared to be part of the state of the buffer, aggregated into its `Instance` region. That is, effects on the array are to be treated as effects on the `BoundedFIFO`. This is only possible because we identify the array as unaliased, which can be verified through analysis [8]. The consequence of these annotations, once verified, is that all state associated with a `BoundedFIFO` instance is protected when a suitable lock is held, including the array contents. (In a later section, we indicate how we document the association of a lock with this state.)

In general, regions of a uniquely referenced object may be aggregated into regions of the referring object by appending the annotation

```
/*@ { src1 in dest1, ..., srcn in destn } */
```

to a reference field annotated to be unique, where region `srci` of the referenced object is aggregated into region `desti` of the referring object.

3.1.3 Aggregation through parameterization

A second kind of aggregation is achieved by parameterizing class definitions by regions. We use a notation similar to C++ templates. Figure 3 shows a pair of classes from the Jigsaw³ open-source Java web server with annotations. These express the intent that the backbone of the linked list referenced by `ThreadCache.freelist` be treated as a single named abstract entity even though its implementation is distributed over many objects: the `freelist` and `freetail` fields of one `ThreadCache` and the `next` and `prev` fields of the many linked `CachedThreads`. The backbone abstraction is made explicit by aggregating all these fields into a single region `Threads` of a `ThreadCache` object. This region is declared on the line labeled 1. The head and tail pointers, `freelist` and `freetail`, are declared on line 3 to be subregions of `Threads`.

To perform the intended aggregation, there must be an identification for each `CachedThread` of which region is to contain its fields `next` and `prev`. The *region parameter* `Backbone`, declared in angle brackets on line 7, accomplishes this. Fields `next` and `prev` are made children of the region bound to `Backbone` on line 8. The parent region (bound to `Backbone`) is specified when a `CachedThread` object is instantiated. This binding conceptually occurs on line 5 when a new `CachedThread` instance is created: the constructor call includes the target `this.Threads`. Thus `CachedThreads` created by different `ThreadCache` objects are parameterized by different regions.

As in other polymorphic systems, all uses of the class name `CachedThread` specify a value for the region parameter (specified by a target); see lines 2, 4, 6, and 8. The declaration on line 8 ensures that a `CachedThread` can refer only to other `CachedThreads` parameterized by the same region.

The parameterization of `CachedThread` enables enforceable separation between distinct `ThreadCache` representations and also enables effects analysis to be less conservative with respect to the uses of `next` and `prev` fields within the implementation of `ThreadCache` because the aggregation adds structure to the region hierarchy.

³Jigsaw is Copyright ©1995–2001 World Wide Web Consortium (MIT, INRIA, Keio Univ.). All Rights Reserved.

```

public class ThreadCache {
  /*@ public region Threads in Instance // 1
  /*@ public region CacheInfo in Instance
  protected int threadcount, usedthreads; /*@ in CacheInfo
  // [code omitted]
  protected CachedThread /*@<this.Threads>*/ // 2
    freelist, freetail; /*@ in Threads // 3

  /*@ lock CacheLock is this protects Instance // A

  private synchronized
  CachedThread /*@<this.Threads>*/ createThread() { ... // 4
  return new CachedThread /*@<this.Threads>*/ (this,...); // 5
  }

  /*@ writes t.ThreadInfo, this.Instance; reads nothing //fx
  synchronized boolean
  isFree(CachedThread /*@<this.Threads>*/ t, ... ) { // 6
    if(!t.isTerminated()) { ... }
    else { ...
      t.prev = freetail;
      if(freetail != null) freetail.next = t;
      freetail = t;
      if(freelist == null) freelist = t;
      usedthreads--; ...
    } ...
  } // [rest of class omitted]
}

class CachedThread /*@<region Backbone>*/ extends Thread { // 7
  /*@ public region ThreadInfo
  private final ThreadCache cache;
  private boolean alive; /*@ in ThreadInfo
  private Runnable runner; /*@ in ThreadInfo
  // [code omitted]
  CachedThread /*@<Backbone>*/ next, prev; /*@ in Backbone // 8

  /*@ lock ThreadLock is this protects ThreadInfo // B

  /*@ writes ThreadInfo; reads nothing
  synchronized boolean isTerminated() { ... }

  // [Additional synchronized methods follow]
}

```

Figure 3: Annotated versions of ThreadCache and CachedThread.

In Section 3.3, we show how to document the association of a lock with this aggregate region. Once this association is documented, the combined analyses (primarily effects and aliasing, as noted above) can provide assurance that the critical shared state is accessed only when the correct locks are held. In general, a class is parameterized by adding the annotation

```
/*@ <region region1, ..., region regionn> */
```

to a class definition. Uses of the class are parameterized by appending `/*@<target1, ..., targetn>*/` to the use.

3.2 Effects Systems

Effects systems answer questions concerning *what state is affected* [20]. These questions must be answered, for example, to ensure all accesses to shared state are identified so that locking policies will be complied with. The object-oriented effects system we use includes effects annotations and composable analyses [22]. As noted above, our approach to concurrency relies closely on this previous work. In summary, the effects system distinguishes two effects on regions: read and write, where writing includes the possibility of reading. Our effects annotations specify a *superset* of the effects a program segment may have on program state—in terms of regions identified by targets—and enables comparison of effects to determine if they *conflict*: if at least one effect is a write and they affect potentially overlapping targets. We use effects to determine which segment of shared state is being accessed, to make verification of uniqueness annotations composable, and to support program transformation.

Method signatures are annotated with a declaration of the regions (identified by targets) permitted to be read and

written as a result of executing the method. See `isFree`, line `fx`, in Figure 3. To preserve abstraction, an effects declaration cannot refer to regions that are less visible than the method being annotated. To enable modular reasoning, a reimplementing of a method in a subclass must conform to the effects declaration in the superclass. There is a subtlety: because subclasses can add fields to existing regions, it is possible for a method in a subclass to affect more state than the declaration in the superclass might otherwise seem to allow.

3.3 Describing Lock–State Association

We now address the questions *what state is shared, and how is access to it synchronized*. Regions are treated like Hoare’s *resources* [23]: as abstract groups of shared state. The programmer identifies a shared region and describes how it is to be protected by introducing a lock annotation into a class:

```
/*@ lock mutex is field protects region */
```

This declares that *region* may be shared and that access to it is mediated via the lock referenced by *field*. We require *field* to be either the receiver (`this` in Java) or an immutable field we select to be the canonical reference to our lock object. (Immutability prevents the identity of the lock from changing.) The lock is known by the abstract name *mutex*, hiding the lock’s representation from clients. If a region is visible in a subclass, any lock requirement associated with it is binding on the subclass. The obvious caveat is that the canonical reference to the lock must also be accessible. Static analysis based on effects can use lock annotations to enforce data access.⁴

Documenting the association between locks and specific state is not a new idea. The novelty of our technique is the association of locks with abstract aggregations of state, with the following benefits: (1) the ability to describe locks that protect state across many objects and (2) direct support for subclassing enabled by the extensible nature of regions. In addition, associating locks with hierarchical regions enables program transformations that can systematically alter the granularity of data protection by splitting across subregions (lock-splitting), or vice versa (lock-merging).

Returning to the class `BoundedFIFO`, we can record how the aggregated state of an instance—the region `Instance` with the array elements—is expected to be protected: by locking on the instance itself. We name the lock `BufLock`:

```
/*@ lock BufLock is this protects Instance */
```

In the `ThreadCache` example, the lock annotation on line A of Figure 3 declares that the region `Instance` of `ThreadCache` objects is protected by the object itself. Because `Threads` is a subregion of `Instance`, the backbone of the linked list is protected by the `ThreadCache` instance. If this deviation were not made explicit, a future maintainer of `ThreadCache` might instead access `next` and `prev` fields by locking their associated `CachedThread` object instead of the appropriate `ThreadCache`, enabling a race condition that could corrupt the list of threads. The annotation on line B in `CachedThread`, in fact, declares that locking a `CachedThread` object protects that object’s `ThreadInfo` region, which does not (and cannot) contain the `next` and `prev` fields.

⁴It is unfortunately not possible to detect shared usage of state *not intended to be shared* without additional knowledge about the existence of threads. This is also true for similar tools and analyses, e.g., RACEFREEJAVA and ESC/Java.

3.4 Lock Usage Annotations

A method is usually assumed to be responsible for introducing necessary critical sections. This common Java assumption is often relaxed when a `private` method assumes its callers already hold locks. The Javadoc for method `java.lang.StringBuffer.copy`, for example, states “[This method] should only be called from a `synchronized` method.”

We answer the question *which locks must be held before a method may be invoked* via an annotation requiring a method to be called from within specific critical sections:

```
/*@ requires mutex1, ..., mutexn */
```

Callers of such a method must acquire the named locks before invoking it. To preserve modular reasoning, the “requires” clause of a method may not be extended by a reimplementation of the method.

We document the requirement that clients of `BoundedFIFO` must perform appropriate locking. Methods that affect shared state are annotated

```
/*@ requires BufLock */
```

Without this annotation, an analysis local to `BoundedFIFO` would warn of potential race conditions because the method implementations do not acquire the appropriate locks, and there would be no way to guarantee that all the call sites follow the convention. With the annotation, clients are required to assure this precondition. Figure 5 shows a fully annotated `BoundedFIFO`.

Locks are objects in their own right, and can be method return values. The method `java.awt.Component.getTreeLock` returns a lock, for example. This is generally specified using

```
/*@ returns lock mutex */
```

Occasionally it is necessary to reorganize code in the service of annotation. For example, if a canonical reference were lacking to a `BoundedFIFO` (e.g., if `fifo` were not final in `FIFOClient`) then we would have to add a new method

```
/*@ returns lock BufLock
public Object getLock { returns this; }
```

Clients would then use `getLock` instead of `fifo` when synchronizing access to the buffer; e.g., `synchronized(fifo.getLock()) {...}`.

3.5 Concurrency Policy

The concurrency policy of a class implementation specifies which methods have potential executions that can be safely interleaved. Policy can potentially be deduced from a combination of representation invariants, formal pre- and postconditions, and the code. In the absence of these potentially expensive specifications, the programmer must express explicitly the design intent that differentiates undesirable concurrency (race conditions, invariant breakers) from desirable concurrency.

In the simplest formulation, policy is expressed as a symmetric boolean matrix indexed by method names. The potential combinatorial challenge can be mediated using several possible techniques, particularly (1) indexing the matrix by regions or effects, and deriving the method matrix from this, and (2) indexing the matrix by sets of related methods (e.g., getters and setters), with obvious derivation of the full matrix. There is a natural tradeoff between succinctness and expressiveness of these policy descriptions. For example, the matrix could be indexed by arbitrary code segments of a class, which would provide perhaps finer-grained interleaving at the expense of more costly policy expression. These options are briefly described in Section 3.5.3. We

	<code>get</code>	<code>put</code>	<code>InfoMethods</code>
(a)	<code>get</code>	×	
	<code>put</code>	×	×
	<code>InfoMethods</code>	S	S

	<code>getter</code>	<code>putter</code>	<code>length</code>	...
(b)	<code>getter</code>	×		
	<code>putter</code>	×	×	
	<code>length</code>	S	S	S
	⋮	⋮	⋮	⋮

Figure 4: The guiding concurrency policies for (a) `BoundedFIFO`, and (b) `FIFOClient`. An “S” indicates allowable safe (invariant-preserving) interleaving; “×” that interleaving is unsafe and disallowed.

distinguish between two uses of concurrency policy: guiding policy that restricts the *implementation* of a class, and client policy that restricts the *use* of particular implementation.

3.5.1 Guiding concurrency policy

The *guiding concurrency policy* of a class sets an upper bound on the extent of interleaving for the methods of a class and its subclasses. In other words, the guiding concurrency policy defines (safe vs. unsafe) concurrency for a class implementation. Guiding policy expresses constraints but not the mechanisms used to prevent interleaving, which may be resource- or client-side. The following coordination techniques are all equally valid, for example: lock-based critical sections, thread-local usage, immutable encapsulations, enforcement of object protocols, and avoiding the need for sharing through deep copying. A pair of methods may have both safe and unsafe interleavings. A method-level boolean formulation of guiding policy cannot capture this distinction, and thus all such interleavings must be disallowed.

An implementation can be checked to verify that the guiding concurrency policy is respected. The checks vary based on the coordination techniques used. For example, an analysis based on locks actually acquired by the implementation (and assumed to be acquired by clients) can provide assurance that certain interleavings cannot occur.

In the absence of representation invariants, guiding concurrency policy is a design decision and so must be trusted. While techniques such as those of [33, 24] may help verify the appropriateness of a guiding policy with respect to explicit representation invariants, our experience in multiple case studies is that informal reasoning (such as performed below) is often sufficient. If the design intent captured by the guiding policy is later determined to be wrong—by allowing a bad interleaving—our assurance framework (see Section 5) facilitates understanding and revision.

The guiding concurrency policy is captured in a separate design document linked with the class it describes. Figure 4a shows the guiding concurrency policy for the `BoundedFIFO`. To reduce combinatorics, the methods `getMaxSize`, `isFull`, `wasFull`, `wasEmpty`, and `length` have been aggregated into the set `InfoMethods` because they all have the same policy properties. The guiding policy prevents, for example, methods `get` and `put` from having interleaved executions with each other or themselves because the value of `numElts` could become corrupted. The “S” for `InfoMethods` × `InfoMethods` means that any pair in the Cartesian product is safe. The (partial) guiding concurrency policy of our example client

FIFOClient is in Figure 4b. The `getter` and `putter` methods are not allowed to interleave: if they did, `getter` could miss a notification and its thread could wait forever.

3.5.2 Client policy

The responsibility for preventing unsafe interleavings may be shared among a resource and its clients. That is, the implementation of the resource may permit concurrency that appears “unsafe” with respect to the guiding policy as long as clients can be assured not to exploit it. The *client policy* of a class specifies pairs of methods that *clients* of the class are (and are not) allowed to invoke concurrently, constraining the design decisions of clients. Adding the following annotation to the implementation of a method m

```
/*@ safe with method1, ..., methodn */
```

declares that methods m , $method_1$, ..., $method_n$ may be invoked concurrently by clients. The method aggregation annotation

```
/*@ letset mset = method1, ..., methodn */
```

reduces the combinatorial pain. To be compatible with inheritance, method pairs not mentioned are assumed to be *unsafe*: existing assurances are not compromised because unknown methods of subclasses are assumed to interfere with known methods. Subclasses may declare newly introduced methods to be “safe with” inherited methods and may also redeclare as safe method pairs previously asserted to be interfering.

The client policy thus describes to clients of a class which potentially unsafe method interactions they avoid. For example, a class implemented as a monitor takes full responsibility for protecting itself and thus has a client policy declaring all pairs of methods as safe. Client policy is a contract, in that its safety guarantees need to persist as a class evolves and is subclassed. From the standpoint of design, this implies that the design of annotations must not be predicated entirely on the particulars of an implementation, but also on potential evolution trajectories. The client policy clearly must not be more permissive than the guiding policy.

A potential race condition exists if a conservative analysis cannot assure consistent regard for the policy, i.e., that unsafe method pairs are not used concurrently by a client. The exact analyses necessary vary with available coordination techniques. Currently we are developing an analysis based on tracking locks. Our relatively strong restrictions on lock references make this an easier analysis than general aliasing, for example. (We have encountered only one example of production code that does not respect these restrictions.)

For the purposes of analyzing a single client, the client policy alone defines the contract for assuring correct usage—there is no need to examine the guiding policy or other annotations on the resource. In the absence of overall guidance on how the client policy is to be enforced, two different clients—separately assured—may use different and incompatible techniques. A system that simultaneously used *both* clients could be unsafe even though both appear to be individually assured safe. Locking annotations and uniqueness annotations ameliorate this composability problem. “Requires” preconditions help document design intent of how the client policy is to be met. If an object whose client is responsible for enforcing policy is uniquely referenced by its (necessarily sole) client, then the client may use any technique to enforce the object’s concurrency policy.

```
public class BoundedFIFO { ...
  /*@unique*/ LoggingEvent[] buf; /*@ {} in Instance)

  /*@ lock BufLock is this protects Instance

  /*@ letset InfoMethods = getMaxSize, length, *
    * wasEmpty, wasFull, isFull */

  /*@ requires BufLock
  /*@ writes this.Instance; reads nothing
  /*@ safe with InfoMethods
  public LoggingEvent get() { ... }

  /*@ requires BufLock
  /*@ writes this.Instance; reads nothing
  /*@ safe with InfoMethods
  public void put(LoggingEvent o) { ... }

  /*@ requires BufLock
  /*@ writes nothing; reads this.Instance
  /*@ safe with InfoMethods
  public int getMaxSize() { ... }

  /* length, wasEmpty, wasFull, and isFull *
   * are annotated like getMaxSize */
}
```

Figure 5: Annotated class BoundedFIFO.

```
public class FIFOClient { ...
  private final BoundedFIFO fifo = ...;
  ...
  /*@ letset WrappedFIFO = putter, getter, length

  /*@ safe with WrappedFIFO
  public void putter(LoggingEvent e) { ... }

  /*@ safe with WrappedFIFO
  public LoggingEvent getter() { ... }

  /*@ safe with WrappedFIFO
  public int length() { ... }
}
```

Figure 6: Annotated class FIFOClient.

In the case of `BoundedFIFO` and its client `FIFOClient`, Figures 5 and 6 show the *client policy* annotations, among others. The convention of `BoundedFIFO` is that clients are responsible for coordination; the client policy of `BoundedFIFO` is therefore identical to its guiding policy. On the other hand, the client policy for `FIFOClient` (see Figure 6) is fully permissive. This reflects the implementation, in which synchronization on `fifo`—in compliance with the client policy and locking annotations of `BoundedFIFO`—already prevents the proscribed interleavings of *both* `FIFOClient` and `BoundedFIFO`. Class `FIFOClient` guarantees this to its own clients through “safe with” annotations. We note, however, that while `length` is safe with `putter` and `getter`, the sensible use of `length` in conjunction with either `getter` or `putter` is in the domain of the concurrency policies of the clients of `FIFOClient`, who otherwise have no worries about the thread-safety of the `FIFOClient`.

3.5.3 A policy mismatch error

We now give an example from Log4j in which a client implementation violates the client policy of a provider. This “policy mismatch” creates several sources of null-pointer and out-of-bounds exceptions, as well as enabling a violation of an internal object protocol [21]. These exceptions can cause the program being logged to terminate prematurely or portions of the logging functionality to misbehave.

Logging-event sources implement the `AppenderAttachable` interface, shown in Figure 7. The functionality of managing event listeners is implemented by the class `AppenderAttachableImpl`, shown in Figure 8. The `AppenderAttachable` interface is implemented by `Category` and `AsyncAppender` (neither shown), which delegate calls to an unaliased `AppenderAttachableImpl` instance, whose method implementations do

```

public interface AppenderAttachable {
    /** Add an appender. */
    public void addAppender(Appender newAppender);

    /** Get all previously added appenders as an Enumeration. */
    public Enumeration getAllAppenders();

    /** Get an appender by name. */
    public Appender getAppender(String name);

    /** Remove all previously added appenders. */
    void removeAllAppenders();

    /** Remove the appender passed as parameter from the list
     * of appenders. */
    void removeAppender(Appender appender);

    /** Remove the appender with the name passed as parameter from
     * the list of appenders. */
    void removeAppender(String name);
}

```

Figure 7: Log4j’s AppenderAttachable interface.

```

public class AppenderAttachableImpl implements AppenderAttachable {
    protected Vector appenderList;

    public void addAppender(Appender newAppender) {
        if(newAppender == null) return;
        if(appenderList == null) appenderList = new Vector(1);
        if(!appenderList.contains(newAppender)) {
            appenderList.addElement(newAppender); }
    }

    /** Call the doAppend method on all attached appenders. */
    public int appendLoopOnAppenders(LoggingEvent event) {
        int size = 0;
        Appender appender;

        if(appenderList != null) {
            size = appenderList.size();
            for(int i = 0; i < size; i++) {
                appender = (Appender) appenderList.elementAt(i);
                appender.doAppend(event);
            }
        }
        return size;
    }

    public void removeAppender(Appender appender) {
        if(appender == null || appenderList == null) return;
        appenderList.removeElement(appender);
    }
    ...
}

```

Figure 8: A partial implementation of AppenderAttachableImpl.

not use any synchronization. Unlike BoundedFIFO, there is no discernible convention on client synchronization.

In Category and AsyncAppender, the methods addAppender, removeAppender(String), removeAppender(Appender), and removeAllAppenders are synchronized. The uses of the method appendLoopOnAppenders are synchronized within Category, but not in AsyncAppender. This synchronization is not sufficient to prevent null-pointer and out-of-bounds exceptions during concurrent use, as will become apparent when we examine the client concurrency policy we inferred for AppenderAttachableImpl. First we infer a correct client policy, then we illustrate how it is violated, and finally we suggest how analyses might detect the policy violations.

3.5.3.1 Client policies for AppenderAttachableImpl.

Generally speaking, the simplest client policy is a monitor-like policy that does not allow the client to execute any pair of methods concurrently. This is concisely recorded using the region-based policy shown in Figure 9a: any two methods that access region Instance of the same AppenderAttachableImpl object are prevented from executing concurrently. An example of a policy that allows more concurrency in the usage of the class is a reader–writer policy. This is most naturally recorded using an effects-based policy de-

	Instance
Instance	×

(a)

	read Instance	write Instance
read Instance	S	
write Instance	×	×

(b)

Figure 9: (a) Monitor-like and (b) reader–writer client policies for AppenderAttachableImpl expressed using regions and effects, respectively.

	(1)	(2)	(3)	(4)	(5)	(6)	(7)
(1) addAppender	×						
(2) appendLoopOnAppender	S	S					
(3) getAllAppenders	S	S	S				
(4) getAppender	S	S	S	S			
(5) removeAllAppenders	×	×	×	×	×		
(6) removeAppender(Appender)	S	×	S*	×	×	S*	
(7) removeAppender(String)	S	×	S*	×	×	×	×

Figure 10: The most permissible client concurrency policy for AppenderAttachableImpl. A bold “S” indicates differences from the reader–writer policy. Pairs marked * are safe due to the underlying Vector.

scription, shown in Figure 9b, differentiating methods based on their effects on a AppenderAttachableImpl object. (Both of these policies could also be more verbosely expressed using method sets or a purely method-based policy matrix.) In the general case, region- and effects-based policy descriptions have the advantage that they are easy and concise to express and they often can be automatically produced based on an analysis of effects.

Returning to our example, the most permissive client concurrency policy for AppenderAttachableImpl is shown in Figure 10. This policy is the same as the guiding concurrency policy, and is more permissive than the reader–writer policy; differences from the reader–writer policy are in bold. Three of the allowable interleavings, marked by asterisks, are safe because of the client policy of the underlying representation used by the implementation, a Vector: its annotations, were the JDK appropriate annotated, indicate that any one of its methods is “safe with” any other. An analysis informed by the effects of AppenderAttachableImpl and the client policy of Vector could deduce the safety of these methods. The other five safe interleavings require reasoning about changes to the fields of the object itself. Let us review some of the reasoning for allowing and disallowing interleaving with method addAppender; see Figure 8.

We allow addAppender × appendLoopOnAppenders. In developing this policy we consider two interactions. (1) addAppender can assign a new Vector to appenderList if there is no vector. But appendLoopOnAppenders terminates when appenderList is null. (2) addAppender can add to the vector once it is present. But appendLoopOnAppenders invokes appenderList.size before its loop, so iterations will stop before any newly added element is reached. In both cases, the interleaved methods execute as if appendLoopOnAppenders executed before addAppender. Thus this pair can be marked as safe in the client policy.

We disallow addAppender × addAppender. Note that both executions could observe appenderList to be null, in which case both would assign a new Vector to the field. One of the added appenders will be lost.

3.5.3.2 Detecting the mismatch.

Failure to use enough synchronization within Category and

`AsyncAppender` enables many interleavings disallowed by this policy and could thus cause unwanted exceptions.

A static analysis can detect the policy violations if the following annotations are added (1) a client policy for `AppenderAttachableImpl`, (2) uniqueness of the delegate references in `Category` and `AsyncAppender`, (3) uniqueness of the `Vector` reference in `AppenderAttachableImpl`, and aggregation of it into the object state.

We omit details of how the policy mismatch can be fixed. We do note, however that it is nontrivial to implement a general-purpose client that is not more conservative than the policy in Figure 10. But there is likely to be little benefit, because the overhead of synchronization is likely to outweigh the benefits of concurrency. However, such a policy can be useful for special-purpose clients, for example, if it is known that there will be a client that only uses the methods `addAppender` and `appendLoopOnAppenders`.

4. RELATED WORK

Extended Static Checking for Java (ESC/Java) [13, 28] is a theorem-prover-based approach for verifying general properties of Java programs. Programs, together with annotations from a subset of the Java Modeling Language [26], are compiled into verification conditions to be proved. Unlike our approach, which avoids expressing representation invariants, the programmer must generally explicitly state object invariants, along with pre- and postconditions. Regarding effects, our system distinguishes both read and write effects and enforces declared effects as an *upper* bound on the effects of methods, whereas ESC/Java is primarily concerned with write effects and does not enforce effects declarations. Leino’s “data groups” [27] aggregate fields of the *same* object for abstraction purposes, but has not been incorporated into ESC/Java.

Both ESC/Java and RACEFREEJAVA [17] associate fields directly with locks; we associate locks with abstract regions, enabling retention of encapsulation and support for program evolution and subclassing. ESC/Java incorporates locking information into the verification condition. RACEFREEJAVA statically checks for unsynchronized access to state using a type-based approach [15, 16]. While [16] describes using existential types in a manner similar to our “returns lock” annotation, RACEFREEJAVA does not implement this feature because it requires resolving aliasing among objects. We avoid this problem by identifying canonical lock references. Classes in RACEFREEJAVA may be parameterized by locks, which is similar to our state aggregation, but the system has no formal model of object state. RACEFREEJAVA supports thread-local classes which are not currently expressible in our system. RACEFREEJAVA has been extended [18] to handle additional programming patterns found to be common sources of false alarms. New annotations include, for example, an annotation indicating the tool should act as if the object’s lock is held while the constructor is executing. Neither ESC/Java nor RACEFREEJAVA can represent unique pointers.

In Guava [3], a dialect of Java without a general synchronization construct, classes belong to one of three categories: sharable monitors, sharable deep-copied values, and unsharable objects. An ownership model [11] prevents sharing of objects: each object has exactly one owning monitor or value, and cannot change its owner. Ownership is statically enforced via an object-granularity effects system and

constraints on alias generation (similar to [8]).

Boyapati and Rinard describe a type-system to enforce locking conventions in Java [7]. It is distinguished by enabling classes to be generic in their protection mechanisms, which are specified when instances are created. Protection is based on object ownership: every object has exactly one fixed owner that is specified through parameterization. Objects representing owners may be given as method preconditions. Before accessing a field of an object or invoking a method with an object precondition, the lock on the object at the root of the ownership hierarchy of the object must be held. Thread-local, unique, and immutable objects (cases that do not require synchronization) are handled using special `thisThread`, `unique`, and `readonly` owners, respectively. An object owned by `unique` must always be unaliased; our system is instead concerned with unaliased *variables*, which may only hold unique references. Uniqueness and immutability are enforced through a simple effects system that records aliasing and write effects that *do not* occur over specific method parameters or local variables.

Object ownership models transitively aggregate the state of an object into the state of its owner. Object ownership allows protection only at the granularity of objects. The state of an object cannot be split across multiple owners, and thus, for example, the classes `ThreadCache` and `CachedThread` of Figure 3 in their current form, cannot be expressed in object-ownership-based systems. Our region-based approach instead models what might be called *field ownership* and is not primarily concerned with the objects referenced by fields.

The Vault programming language [12] statically tracks keys (capabilities) associated with fields and variables to enforce resource management, such as locking protocols. Keys can model non-hierarchical regions. The state of an object may be aggregated into multiple other objects by parameterizing types by keys. However, no general state naming system is provided beyond the key-field associations the programmer chooses to make. Vault tracks key aliases, and enforces key uniqueness, but provides no mechanism for ensuring the uniqueness of an object reference. An advantage of Vault is that it does not require lock management to be syntactically scoped, but it provides no structure in the use of keys for locking purposes. The programmer must describe the locking conventions, including the lock acquisition and release mechanisms, using key annotations. It is not possible to describe reentrant locks. Because keys can be used for many purposes (e.g., preventing memory leaks, enforcing object protocols), it cannot be assumed that any particular key defines a field-lock relationship.

The assertion facility proposal for Java [5] can associate locks with threads, but the checking is dynamic.

None of the systems described above supports the description of concurrency policy. Several languages contain synchronization constructs resembling concurrency policy. In Path Pascal [9], class-level path expressions specify both object protocol and concurrency constraints. In [1, 29], methods are annotated with those methods with which their execution may interleave (a.k.a. compatibilities). In CEiffel [29], subclasses may arbitrarily alter the compatibilities of a method, and abstract methods may not specify any compatibilities: we conclude that these annotations are primarily for implementing synchronization rather than for describing policy.

Lucassen extends his effects system with monitor-call ef-

fects “to give the programmer some control over the way in which computations are interleaved” [30]. All monitor-call effects are defined to interfere with each other. Described as a client concurrency policy, the policy would be at the system granularity and all methods of all monitors would exclude each other.

Schwarz and Spector consider transactions on abstract data types [35]. Their technique is similar to our policy in that their motivation is to be able to reason about the interactions of operations on abstract types.

Noble, Holmes, and Potter describe an *algebra of exclusion* [32] for describing method-level exclusion, that is analogous to our client policy. The algebra is intended for reasoning about the behavior of objects, particularly composite objects whose behavior is derived from delegated objects, and in this sense is complementary to our current work. The algebra does not handle inheritance.

5. APPLICATIONS & FUTURE WORK

Our annotation approach is motivated by the need to assure thread safety, particularly as programs undergo evolution. We have suggested throughout how analyses based on effects and our other annotations might be used to check and enforce various concurrency-related properties. The specification and implementation of these analyses are subjects of current research. We intend for the analyses to be part of a tool that also assists with the introduction and management of annotations. The annotations and analysis results are not an end themselves: they are in the service of our larger goals of program assurance and program evolution.

5.1 Assurance and Chains of Evidence

As annotations are added to `BoundedFIFO`, additional conclusions can be drawn in support of various code-safety attributes. Region annotations and effects declarations based on the regions enable a variety of analyses, particularly of uniqueness and locking requirements. These annotations also provide a framework for expressing policy. When we add lock-state associations, the same analyses can verify compliance. These associations can also trigger requirements to add “requires” annotations. At this point, policy can be specified, and compliance can be directly verified using locking information.

A similar account holds for `FIFOClient`, except that once client policy for `BoundedFIFO` is in place simple analyses can assure that `FIFOClient` is compliant with policy and “requires” annotations. These analyses may place requirements on *its* clients.

As the code evolves, additional regions and subregions may be defined, policy may be liberalized, and other changes may occur. Because of the incremental nature of the linking of annotations, code, and analyses, we suggest that the effort required to sustain consistency among these elements is more likely to be proportional to the extent of the change rather than the extent of the overall code base. But proof of this conjecture may be difficult to achieve.

5.2 Program Evolution

Log4j offers an interesting example to illustrate the application of our approach to program evolution. Between versions 1.0.4 and 1.1b1, a `resize` method was added to `BoundedFIFO`. Unlike the original methods, `resize` (code omitted) is `synchronized`. This is compatible with existing an-

notations, but the compatibility is not obvious because all other methods expect the *caller* to do the locking, and with the same object. But `resize` performs the locking itself, and no “requires” annotation is necessary. Clients of `BoundedFIFO` thus must (1) not lock the object before calling `resize` and (2) use the object as the lock for all other methods. If existing clients comply with annotations, the change is benign. If a rogue client had synchronized on some *other* object, then our analyses would detect the race.

More generally, our annotations are intended to support (1) expression of design intent regarding concurrent access to shared data, (2) composable static analyses to establish assurance of consistency of code with design, (3) establishing preconditions for various kinds of restructuring program transformations [34], (4) informal reasoning by programmers concerning design intent for concurrency.

5.3 Additional Annotations

The work presented herein focuses on concurrency design decisions related to integrity of state, and we have not discussed other concurrency-related issues, particularly deadlock. It is well known that annotations specifying and managing partial orders for lock acquisition can assist in assuring deadlock freedom. These annotations are global in character, but can be made incrementally (i.e., by adding new pairs to the partial order relation).

We are presently exploring two additional classes of annotations: (1) blocking-related annotations to assist in reasoning about condition variables, and (2) “thread coloring” annotations to identify associations among threads, data, and code segments [D. F. Sutherland, personal communication]. Such annotations would be useful when reasoning about policy by enabling an analysis to determine that particular segments of code cannot interleave because, for example, they must always be executed by the same thread. Explicit identification of threads also provides a means to incorporate thread-local objects and to detect shared usage of state not intended to be shared.

Defining the scope of a concurrency policy is an additional challenge. Many cases exist where the scope of a policy must go beyond a single class (and its subclasses). For example, classes collaborating via an enumerator may require policy to be expressed at the level of the classes involved: in `AppenderAttachableImpl`, the enumeration-returning method `getAllAppenders` is “safe with” various mutator methods, but the resulting enumeration object is *not* “safe with” those methods. We believe there may be patterns of policy that correspond with patterns of code structure.

Expressing concurrency policy in terms of combinations of regions, effects, and method scopes, as explored in Section 3.5.3, offers a possible approach to this problem. Specifying policy in terms of regions enables the capture of all methods that can affect those regions, and thus captures access paths via enumerations, for example.

The combinatorics of policy expression for class collaborations can be reduced by reasoning about interactions over the (typically small) set of regions defined by a class instead of the (possibly large) set of actual methods.

6. CONCLUSION

We present in this paper notations to express several key aspects of low-level design related to shared-memory concurrency, such as used in Java and Ada 95. We also demon-

strate how these annotations enable assurance of a variety of properties related to safe concurrency, principally relating to respect for representation invariants or, more informally, absence of “race conditions.”

Our design intent in developing these notations and associated analysis techniques is for benefits to accrue incrementally as annotations are added to a system, and for the annotations themselves to be easy for programmers to express—that is, for the annotations to naturally express design commitments already in the minds of programmers. This follows the tradition of other low-level design notations now in general use, such as Hungarian notation and class diagrams.

In order to test our approach, we have applied our approach to production code from the JDK and from a variety of open source Java projects. In these case studies we have been able to document non-trivial concurrency-related design decisions, discover concurrency-related errors, and add to the assurance of “thread safety.”

7. ACKNOWLEDGMENTS

The authors thank John Boyland, Edwin Chan, Thomas Gross, Tim Halloran, Elissa Newman, Orna Raz, Christoph von Praun and the anonymous reviewers for helping us to improve this paper. Effort sponsored in part through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298 and in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL), Air Force Materiel Command, USAF, under agreement number F30602-99-2-0522. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of NASA, DARPA, AFRL, or the U.S. Government.

8. REFERENCES

- [1] G. R. Andrews and J. R. McGraw. Language features for process interaction. In D. B. Wortman, editor, *ACM Conf. on Lang. Design for Reliable Softw.* ACM Press, Mar. 1977.
- [2] Apache Software Foundation. Log4j project. <http://jakarta.apache.org/log4j/docs/index.html>. Current Jan. 2002.
- [3] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *OOPSLA '00*. ACM Press, Oct. 2000.
- [4] J. Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, 2001.
- [5] J. Bloch. Proposed final draft, JSR-41: A simple assertion facility for the Java programming language. <http://jcp.org/jsr/detail/41.jsp>, June 2001. Current Jan. 2002.
- [6] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
- [7] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *OOPSLA '01*. ACM Press, Nov. 2001.
- [8] J. Boyland. Alias burying: Unique variables without destructive reads. *Softw. Pract. and Exper.*, 31(6), May 2001.
- [9] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In E. Gelenbe and C. Kaiser, editors, *Oper. Syst. Symposium*. Springer, Apr. 1974.
- [10] E. C. Chan, J. T. Boyland, and W. L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *ICSE '98*. IEEECS, 1998.
- [11] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA '98*. ACM Press, Oct. 1998.
- [12] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI '01*. ACM Press, May 2001.
- [13] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq SRC, Dec. 1998.
- [14] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *FSE '94*. ACM Press, Dec. 1994.
- [15] C. Flanagan and M. Abadi. Object types against races. In J. C. M. Baeten and S. Maw, editors, *CONCUR '99*. Springer, 1999.
- [16] C. Flanagan and M. Abadi. Types for safe locking. In S. D. Swierstra, editor, *ESOP '99*. Springer, 1999.
- [17] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI '00*. ACM Press, May 2000.
- [18] C. Flanagan and S. N. Freund. Detecting race conditions in large programs. In *PASTE '01*. ACM Press, June 2001.
- [19] D. Garlan, R. T. Monroe, and D. Wile. ACME: An architectural description interchange language. In *CASCON '97*, Nov. 1997.
- [20] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *1986 Conf. on Lisp and Func. Program.* ACM Press, 1986.
- [21] A. Greenhouse. Bug 1507: Improper synchronization in AsyncAppender and Category causes race conditions. <http://nagoya.apache.org/bugzilla/>, Apr. 2001. Current Jan. 2002.
- [22] A. Greenhouse and J. Boyland. An object-oriented effects system. In R. Guerraoui, editor, *ECOOP '99*. Springer, 1999.
- [23] C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrot, editors, *Oper. Syst. Techniques*. Academic Press, 1971.
- [24] L. Lamport. The ‘Hoare logic’ of concurrent programs. *Acta Inf.*, 14(1), 1980.
- [25] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, second edition, 1999.
- [26] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*. Kluwer, 1999.
- [27] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98*. ACM Press, Oct. 1998.
- [28] K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user’s manual. Technical Note 2000-002, Compaq SRC, Oct. 2000.
- [29] K.-P. Löhner. Concurrency annotations for reusable software. *Commun. ACM*, 36(9), Sept. 1993.
- [30] J. M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. PhD thesis, MIT, Sept. 1987.
- [31] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.
- [32] J. Noble, D. Holmes, and J. Potter. Exclusion for composite objects. In *OOPSLA '00*. ACM Press, Oct. 2000.
- [33] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6(4), 1976.
- [34] W. L. Scherlis. Systematic change of data representation: Program manipulations and a case study. In C. Hankin, editor, *ESOP '98*. Springer, 1998.
- [35] P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract types. *ACM Trans. Comput. Syst.*, 2(3), Aug. 1984.
- [36] C. Simonyi and M. Heller. The Hungarian revolution. *Byte*, 16(8), Aug. 1991.