Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 58 (2005) 384–411

Science of Computer Programming

www.elsevier.com/locate/scico

Observations on the assured evolution of concurrent Java programs

Aaron Greenhouse[a,*], T.J. Halloran[b], William L. Scherlis[c]

[a] *Software Engineering Institute, 4500 Fifth Ave, Pittsburgh, PA 15213, United States*
[b] *Air Force Institute of Technology, Department of Electrical & Computer Engineering, 2950 Hobson Way, Wright-Patterson AFB, OH 45433-7765, United States*
[c] *School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213, United States*

Received 1 November 2004; received in revised form 15 January 2005; accepted 1 March 2005
Available online 15 June 2005

Abstract

Evolving and refactoring concurrent Java software can be error-prone, resulting in race conditions and other concurrency difficulties. We suggest that there are two principal causes: concurrency design intent is often not explicit in code and, additionally, consistency of intent and code cannot easily be established through either testing or inspection.

We explore several aspects of this issue in this paper. First, we describe a tool-assisted approach to modeling and assurance for concurrent programs. Second, we give an account of recent case study experience on larger-scale production Java systems. Third, we suggest an approach to scalable co-evolution of code and models that is designed to support working programmers without special training or incentives. Fourth, we propose some concurrency-related refactorings that, with suitable analysis and tool support, can potentially offer assurances of soundness.

Keywords: Java; Concurrency; Program assurance; Static analysis; Program transformation; Refactoring; Program evolution

* Corresponding author.
E-mail addresses: aarong@sei.cmu.edu (A. Greenhouse), thallora@afit.edu (T.J. Halloran), scherlis@cmu.edu (W.L. Scherlis).

0167-6423/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2005.03.002

1. Introduction

Reasoning about concurrent Java programs is a challenge for both programmers and software tool designers. Evolving concurrent programs without introducing subtle bugs can be even a greater challenge. Developers must constantly raise and address questions such as: *What data is shared by multiple threads? Is it accessed safely? What locks should be held when particular portions of shared state are accessed? Whose responsibility is it to acquire the lock? Is this delegate object also protected? If so, is it uniquely referenced by its referring object?*

Two questions. These bread-and-butter development questions are instances of two kinds of general questions. The first question is what is the concurrency-related design intent. Generally speaking, race conditions occur just when representation invariants are violated as a result of simultaneous action by multiple threads. This is difficult to ascertain by inspection, because when representation invariants are not expressed, there is no obvious way to distinguish good concurrency from races and potential data corruption.

In Java, lock objects are used to prevent interference when critical shared state is accessed. This suggests that an identification of portions of shared state may provide a useful surrogate for a representation invariant. Java, however, does not provide a means to express an association between a lock object and the state it is intended to protect. These associations are therefore an important category of concurrency-related design intent.

Developers tend to rely on conventions, for example, that an object is used as a lock for its own state, and takes care of its own locking. These conventions assist a code reader or inspector in inferring unstated design intent. But of course they are not universally applicable and are often breached. Consider, for example, a resource pool. In some implementations, the lock of a pool object protects the state of the pool object itself plus those specific portions of the state of *pooled* objects that represent the backbone of the linked list of the objects in the pool.

Another frequent example is the use of a lock of a referring object to protect the state of uniquely referenced delegate objects, particularly arrays and lists. This is typical in event queue representations. Another issue is responsibility for lock acquisition. There are many cases where a client of a shared object is meant to be responsible for acquiring and releasing locks—this is particularly useful when atomicity constraints require a lock to be held over a sequence of method calls.

Another example of design intent is the decision, typical in GUI frameworks, to follow a single-threaded policy to protect the integrity of shared data without the use of locks. The Java AWT, for example, has rules regarding the appropriate use of its event thread, including rules that restrict which threads may invoke event callback methods.

The second question is whether there is consistency between code and the stated design intent. An example of consistency for a lock-based model is an assurance that the correct lock is *always* acquired prior to accessing shared state [18,15]. For non-lock concurrency, the assurance pertains to the identity of the threads that touch critical state or code [26]. Failures to achieve consistency thus reflect either flaws in the model, flaws in the code, or inadequacy of the verification and analysis tools.

Adoptability and scalability. Addressing these two questions, particularly in the case of fast-paced iterative development environments, can be highly problematic. While programmers may communicate design intent informally among themselves, there has been relatively little success in capturing intent in sufficiently precise representations that tools can be used to assist in assuring consistency. The high "expression cost" often creates an effective adoption barrier, and the well-known difficulties in general-purpose assurance for larger systems create an effective scaling barrier. Generally, the only exceptions are small components of highly critical systems. As a consequence, there is a perception that both the expression of model information and the assurance of its consistency with the code present an overwhelming barrier to providing deep analytic assurance for routine Java programming efforts. The experience of many projects suggests that the barrier involves both usability and scalability. Usability barriers include ease of expression of design intent, ease of use of analysis tools, value of information provided, etc. Scalability barriers include extent of design intent information to be expressed and difficulty of the analysis task as a function of the scale and complexity of the code.

One of the challenges in modeling and reasoning about concurrency is that most interesting concurrency-related properties defy both traditional testing and inspection techniques. There is no single place in the code to look to find either expression of model information or evidence of compliance. In particular, the reasoning process to ensure consistency of code and model is almost always non-local in the structure of the code. This raises a challenge for programmers attempting to accomplish informal reverse engineering—often of their own code—and also for tool developers seeking to assist in evolving and assuring concurrent Java programs.

We present here an approach to concurrency-related assurance and evolution designed to address these challenges of practicability and scale. Our approach is to identify small compromises of inferential and expressive power that yield big improvements in adoptability for working programmers as well as in scalability to components and systems of realistic size.

In the sections below, we present our approach to modeling and reasoning about Java concurrency. As noted, there are specific limits we accept on expressiveness and power to achieve this. In particular, rather than requiring a programmer to express and verify full representation invariants for data structures, we instead substitute a model of "guilt-by-association", in which those constituents of shared state that participate in some notional representation invariant are associated with each other into "regions" [18,16]. Race conditions, by definition, occur when a representation invariant is expected to hold at a place where in fact it does not, due to the interleaved execution of a separate thread. In our approach, we focus on identifying the constituents of state that might be related by a putative invariant, but we avoid elucidating their precise relationship. The hypothesis underlying this approach is that modeling at this more abstract level can provide concrete value in establishing safe concurrency. This has been reinforced through extensive case study experience.

Approach and prior work. Our approach has four elements:

(1) The incremental expression of "mechanical" design intent (what some call "non-functional" requirements) for Java concurrency.

(2) An incremental and composable approach to analysis-based assurance of consistency of that intent with code.
(3) Support for rapid iteration in the co-evolution of code and design intent.
(4) Support for semi-automated refactorings.

Underlying this approach is a tool based on a suite of composable semantic program analyses supported by a complex assertion- and proof-management scheme that supports composition and incrementality.

We have previously described some results that contribute to elements (1) and (2) of our approach above—expressing design intent that enables programmers to capture model information in a way that enables assured consistency between the expressed design intent and code [18,15,26,16,6,8]. In this paper we summarize recent experience regarding these first two elements, and additionally offer some potential directions for addressing elements (3) and (4). This builds on extensive case study experience using our Eclipse-based tool.

Practicability. Our overall approach is designed from the outset to be practicable—feasibly adoptable by real programmers working on deadline. One of the lessons we have learned is that considerations of practicability have a significant influence on every aspect of the approach, including even the design of underlying analysis algorithms—for example, to support component composition, explicit cutpoints, programmer debugging, and interactions among analyses.

There is also a perception, which is also generally well founded, that assurance raises a formidable barrier to evolution of both code and models. Part of the challenge is that in production development efforts consistency is almost always partial, as are the associated models. That is, programmers must be able, incrementally, to accrete model information, to reason about models and code, and to evolve both models and code. This evolution can be entirely manual or it can be tool assisted, as in the semi-automated refactorings implemented in tools such as Eclipse.

Our approach to practicability is based on three key principles:

(1) *Incrementality and early gratification.* Any increment of effort we ask programmers to undertake should yield a generally immediate reward in the form of bug finding, assurance creation, guidance in evolution, or model expression. Our intent: Useful assurances can be obtained with minimal or no annotation effort, and additional increments of annotation yield additional increments of assurance. This is a reason why we have avoided any requirement for explicit expression of representation invariants.
(2) *Familiar expression.* Properties should be expressed tersely and using terminology already familiar to programmers. Generally speaking, we cannot require practicing programmers to master unfamiliar verification formalisms. This is a challenge, because representation invariants underlie the semantic distinction between races and desired concurrency. Our more abstract proxies have proven to be sufficient for diverse case studies.
(3) *Cut points and composability.* We should be able to handle individual components separately from each other, developing composable assurances, which can be linked together to form "chains of evidence" supporting an overall system-level claim. There are two challenges: First, cutpoints need to be expressed in a way that satisfies the first

two principles above. Second, analyses need to be usefully precise, avoiding excessive conservatism, for a broad range of existing Java code.

Refactoring. Rapid development iteration is increasingly a tool-supported activity, with tools used to assist in restructuring or *refactoring* of code. Refactorings are systematic plans for transforming source code, generally in ways that preserve behavior. For example, one kind of refactoring is the extraction of a new abstract superclass from two similar classes—without changing the behavior of the program. This refactoring can eliminate redundancy in the source code and improve the programmer's ability to understand and evolve the program. Another example of a refactoring is extraction of a new method definition from one or more existing method definitions. This can involve subtle reorderings of computation, for example, of the code "left behind" at the call site to calculate values of actual parameters.

Refactoring is a challenge from the standpoint of bug-prevention and program assurance. If one starts with a correct, i.e., assured, program with respect to a set of models, then the refactored program should still be correct after applying the refactoring. This is particularly important when refactoring results in broad structural changes. Both code and models may be transformed. And, of course, the refactorings may have associated soundness preconditions that can be established only through some modeling and analysis.

A use of refactoring is to make programs easier to understand by making intent more self-evident. Programs thus become safer to evolve [14]. There is an unfortunate irony: the refactoring process can itself be unsafe. When soundness preconditions are not identified or cannot be assured, for example, due to a lack of models, both automated and manual refactoring can be risky. Refactorings implemented in tools may be unsound, particularly with respect to concurrency and other non-local program attributes.

Determining whether the program satisfies the preconditions for a refactoring rule may require explicit knowledge of non-local design intent. For example, *What state might be read or written by this method? Is this field aliased? Is this class intended to be subclassed? Who are the clients of this class?* [22,24]. Best practice for manual refactoring is generally considered to require explicit reverse engineering, possibly using programming tools to search program text, and then applying a refactoring either by manually manipulating program text or using a tool [14]. Most automated refactorings will result in compilable code. But this is not sufficient to guarantee that program behavior is preserved.

Indeed, it appears that there is a kind of pragmatic trade-off between soundness and "manipulative power" evident in the present generation of tool-implemented refactorings. If so, increments in our ability to assure soundness of refactorings could have a significant impact on the range and sophistication of transformations available through mainstream code development tools.

1.1. Outline

In this paper we (1) summarize our approach to tool-assisted modeling and assurance for concurrent programs, and (2) provide an account of recent case study experience on larger-scale production Java systems such as jEdit, Log4J, `util.concurrent` (a widely used concurrency library), as well as several commercial and government systems. We then (3)

offer an approach to principled co-evolution of code and models that is intended to meet the practicability criteria above, and (4) propose some concurrency-related refactorings that, on the basis of models and analyses, can be implemented soundly in tools. Because models are integral to our approach, it is important to avoid any requirement for programmers to reinvent models after restructuring code. In our approach to refactoring, therefore, both code and models are manipulated simultaneously. Our intent is to enable development of a more powerful generation of sound refactorings, including refactorings whose purpose is to assist developers in making effective use of concurrency.

2. Our assurance tool

We have implemented a prototype tool, sketched in [17], within the Eclipse IDE. This seemingly benign plug-in embodies the program analysis and assurance techniques described in previous work [18,15,16,6]. It has its own internal representation that supports a variety of views and analyses, detailed below. We have applied this prototype tool to a number of mid-scale production concurrent Java programs, and have had success in recovering and capturing portions of concurrency-related design intent for these systems. The tool provides both bad news and good news: we have uncovered a number of previously unknown concurrency errors, and we have provided analytic assurances regarding consistency of code and models. In this section, we provide a brief overview of the capabilities of our tool, describe programmer interaction with our tool, and report on case study experience with our tool.

2.1. Tool capability

Programmers using our tool record design intent in terms of concepts already familiar to developers. Models of design intent are expressed as program annotations on source code in a format familiar to users of Javadoc. A design goal is for each annotation to provide some immediate value by answering a question about the code. This is a crude incentive system: working programmers should want to introduce annotations because they receive near-immediate benefits that are useful to ongoing development activity—as well as to overall quality assurance of the evolving system.

Our tool allows analysis to proceed in increments across the code base and associated models. An unannotated class is merely a class that has no models against which it can be verified. Unannotated Java code is never treated as wrong; it just lacks claims of consistency with design intent.

There are other tools, including RACEFREEJAVA [12] and Guava [2] that can assist in providing assurance of thread safety. These tools use composable analyses: modular type systems allow a program to be analyzed on a per-class basis. But they have generally not focused on providing the programmer with a usage model based around incremental annotation and early gratification. Our assurance tool is designed to be able to provide useful results with as little as a single annotation; see, for example, Section 2.3.2.

Our annotations for expressing design intent and their associated analyses can be categorized as follows. Appendix A provides a brief introduction to the use of our more common annotations.

- **Aggregations of state.** These declarations enable a programmer to declare abstract hierarchical *regions* of state that can both subdivide and span across objects. The hierarchy allows regions to be wholly nested in other regions providing a vocabulary for referring to state at different granularities. These state models can exploit uniqueness of references to aggregate uniquely referenced objects into the state of other objects [15,16].
- **Effects.** A programmer can declare the upper bounds of the effects of a method—the state it reads and writes—in terms of regions. Analysis can verify that implementations respect the declarations and suggest appropriate declarations for unannotated methods [15,16].
- **Aliasing intent.** These declarations enable a programmer to declare that a field or return value is intended to be unaliased. Parameters that are not aliased by methods can also be declared and verified [6].
- **Locking intent.** These declarations enable programmers to declare models that associate locks with state. Analyses can verify that state is accessed only when the appropriate lock is held [18,15]. These annotations use the models of state provided by regions. In addition, the programmer can declare that a method requires a particular lock to be held by the caller, and can declare that a method returns a particular lock.
- **Concurrency policy.** These declarations enable identification of methods that may be safely executed concurrently [18,15]. In general, the programmer declares which methods have safe interleavings based on their critical sections. We hypothesize that, for lock-based concurrency, concurrency policy combined with models of locking intent is a suitable surrogate for representation invariants.
- **Thread identification.** These declarations provide a way to associate particular threads with code segments and regions of state [26]. We do not elaborate these ideas herein. These declarations provide an approach to the management of the non-lock concurrency typical of GUIs, and support static assurances regarding appropriate use of threads in real-time Java [5].

Our models of design intent have been sufficient to capture the majority of Java concurrent programming idioms we have encountered in a broad sampling of production systems, including both commercial code and widely adopted, high-quality open source code. But these models are not sufficient to capture all concurrency design patterns. For example, we presently do not model and reason about thread-local objects, and we cannot yet describe designs that use arrays of locks or other indirect means of referring to locks.

2.2. Programmer–tool interaction

Fig. 1 shows a portion of the user interface of our prototype Eclipse-based tool. The programmer enters annotations, examples of which are provided below, into code using the normal Eclipse Java editor. As soon as the programmer saves a compilation unit containing annotations, the tool performs build, executes analyses, and displays results. Our tool, similar to the Eclipse Java compiler, is incremental and runs in the background while the programmer continues to work. Thus, the tool unobtrusively monitors model–code consistency as a programmer works on code, and it provides quick feedback as a programmer works to express models. Generally speaking, the analysis run time is a

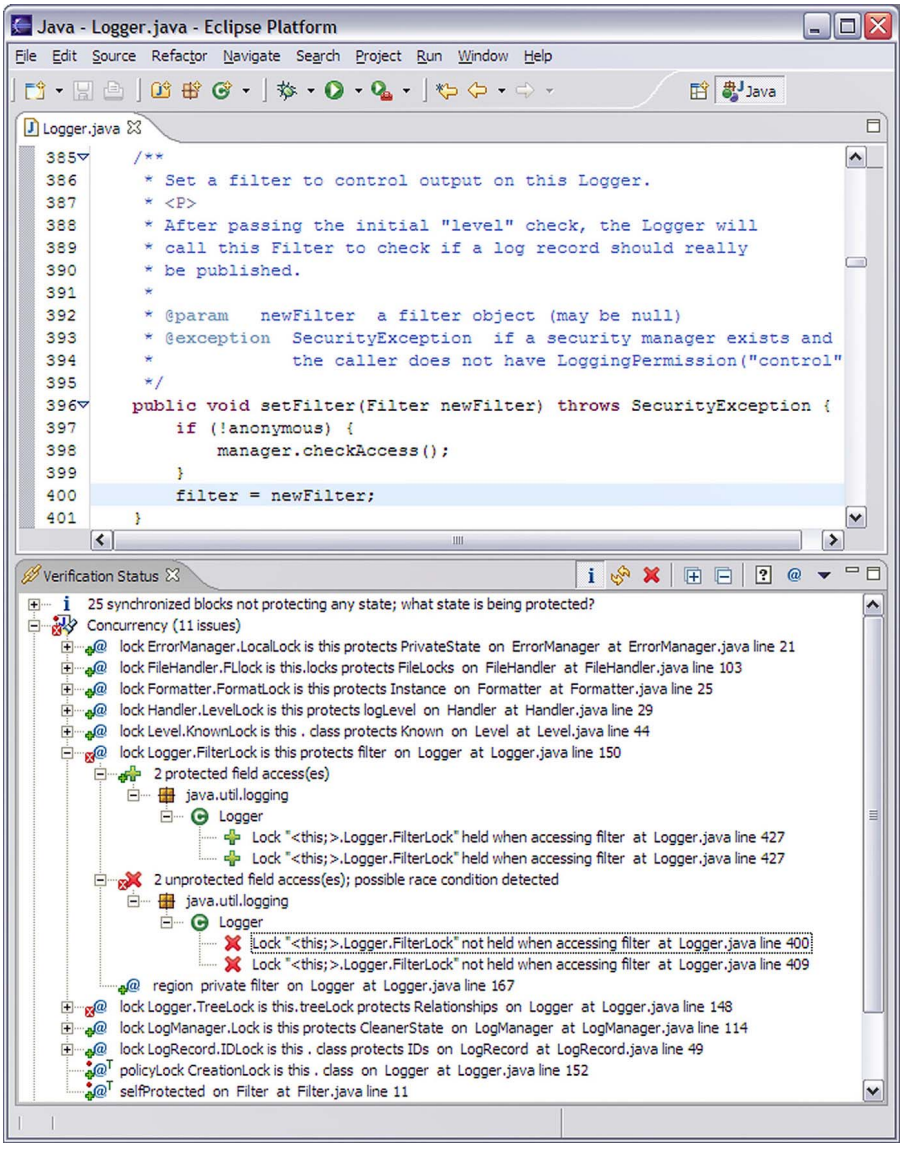

Fig. 1. Our prototype concurrency assurance tool highlighting a race condition within the `java.util.logging.Logger` class (a part of the standard Java library).

function of the number of expressed models and the complexity of the control-flow structure of the code associated with the models.

The results window shown at the bottom of Fig. 1 reports model–code assurance results with a green "+" to indicate consistency and a red "×" to indicate inconsistency. A blue "**i**" highlights potential next steps for the programmer—inferred from the existing code

and models. For example, the top "**i**" result in Fig. 1 highlights 25 uses of locks that have not been explicitly associated, via a model, with the state they are intended to protect. In this case, the hope is that the programmer will eventually investigate the `synchronized` blocks associated with the messages and declare the necessary locking models. Because these "**i**" results, like any inference not based upon explicit design intent, are subject to false positives, they can be filtered out by toggling the blue "**i**" button in the upper right of the "Verification Status" window. The smaller "+" and "×" icons at the lower left corner of each item indicate the overall status of verification with respect to that model.

Our tool also highlights any assertion not fully verified by program analysis by placing a small "red-dot" in the upper right corner of that result and any result dependent upon that result. For example, the bottom two results, policy lock `CreationLock` and the "self protected" assertion on `Filter`, are trusted—indicated by the small "T"—causing a "red dot" to appear on those results and the overall folder for concurrency results. This enables incremental progress: programmers can deliver results contingent on analyses not yet done and on code not yet written.

2.3. Tool experience

We have applied our tool to a number of concurrent Java programs from established open source projects as well as to (larger) industry and government systems. We elaborate four examples below: `java.util.logging`, the Apache Jakarta Log4j library, the open source text editor jEdit,[1] and the well-known concurrency utilities package `util.concurrent`.[2] In most of the systems we examined, we both uncovered race conditions and obtained many positive results. The positive results are in the form of captured design intent coupled with analyses that verify consistency of code with the models.

The size of each program discussed below and the number of annotations we added within its source code are shown below:

System	kSLOC	Annotations
jEdit v4.1	72.3	36
log4j v1.2.8	19.8	43
util.concurrent v1.3.2	10.3	158
util.logging v1.4.1_01	2.3	45
sponsor program	7.4	12

Our first example shows the kinds of errors a well-intentioned developer can make when evolving a class with a non-trivial, unstated concurrency model.

2.3.1. Case study: Log4j's `BoundedFIFO`

We introduce our technique here using the `BoundedFIFO` class, which implements a shared buffer between two threads, taken from the Apache Jakarta Log4j source code.[3] Log4j is a widely adopted library for event logging; its capabilities are similar to those of

[1] http://www.jedit.org/. Bugs 893519 and 893735.

[2] http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html.

[3] http://logging.apache.org/log4j/docs/index.html. Log4j is ©1999 Apache Software Foundation.

its SDK successor `java.util.logging`. Consider the annotated and elided code fragment below:

```
1 /** @lock BufLock is this protects Instance */
2 public class BoundedFIFO { ...
3   /** @aggregate [] into Instance
4    *  @unshared */
5   LoggingEvent[] buf;
6   int numElts = 0, first = 0, next = 0, size;
7
8   /** @singleThreaded
9    *  @borrowed this */
10  public BoundedFIFO(int size) {...}
11
12  /** @requiresLock BufLock */
13  public LoggingEvent get() {...}
14
15  /** @requiresLock BufLock */
16  public void put(LoggingEvent o) {...}
17
18  public synchronized void resize(int newSize) {...}
19 }
```

Recall that Java has no way to express the association of locks with shared state, though there are conventions. The `@lock` annotation on line 1 declares that the `BoundedFIFO` object instance, `this`, is intended to protect all the object's fields. Specifically, this `@lock` annotation declares that the region `Instance` is protected by the object referenced by `this`. The lock is given a name, `BufLock`, because the object may otherwise be anonymous. The region `Instance` is a default region declared in `Object` and is automatically populated with the instance fields. Fields may alternatively have an explicit parent region declared via `@mapInto`; see Section 2.3.4 for an example.

The `@requiresLock` annotation on lines 12 and 15 indicates that holding a lock on the `BoundedFIFO` object (e.g., `synchronized (fifo) { e = fifo.get(); }`) is intended to be a prerequisite for callers invoking these methods. The contents of the buffer are actually a separate object, the array `buf`. Lines 3 and 4 aggregate the elements of `buf` into the state of the `BoundedFIFO` object instance. They also declare that references to the array are not "leaked" to other objects within the program—that the only references are from within the `BoundedFIFO` object. Leakage may also occur from constructors. Line 8 declares that during construction, the new object is only accessed by a single thread, i.e., the one that invoked the constructor. Knowing this, our tool does not have to enforce the use of a locking protocol within the constructor's implementation. Finally, line 9 declares that the constructor does not "leak" a reference to the object itself; this is used to assure the consistency of the `@singleThreaded` annotation. On the basis of this model, our tool provides an assurance of consistency of code and model. Multiple program analyses contribute to this, including binding, typing, unique references (a specialized alias analysis), may-equal (another specialized alias analysis), effects, and our special-purpose lock analysis.

Between versions 1.0.4 and 1.1b1 of Log4j, a `resize()` method, declared on line 18 above, was added to enable resizing of the buffer. Unlike the other methods in the class, `resize()` is `synchronized`. Our tool assures the safety of this new method without the need

for additional annotation. Introducing this method also led to a policy failure relating to the use a wait–notify protocol. See our reports in Apache bugs 1505, 1507, 23912, and 26224.

2.3.2. Case study: `java.util.logging.Logger`

The class `java.util.logging.Logger` was introduced in Java 2 SDK version 1.4. It is intended to be safe for use in multi-threaded programs: the Javadoc claims "All methods on Logger are multi-thread safe". We introduced a single annotation to express a portion of the locking model.

```
1 /** @lock FilterLock is this protects filter */
2 public class Logger { ...
3   private Filter filter;
4
5   public void setFilter(Filter newFilter)
6   throws SecurityException {
7     if (!anonymous) manager.checkAccess();
8     filter = newFilter;
9   }
10
11  public void log(LogRecord record) { ...
12    synchronized (this) {
13      if (filter != null && !filter.isLoggable(record))
14        return;
15    } ...
16  }
17 }
```

The tool is unable to establish assurance because method `setFilter` does not acquire `FilterLock` before writing to field `filter` on line 8. This enables a race with the `log` method in which `log` checks that `filter` is non-`null` (line 13), `setFilter` writes `null` to `filter` and then `log` dereferences the now-`null` `filter` (also line 13) resulting in an exception. This example highlights the non-local character of concurrent programming that makes it so difficult to debug: even though method `log` is written correctly, it is compromised by the incorrect implementation of `setFilter`. The race can be fixed by enclosing the assignment to `filter` at line 8 above within a block `synchronized` on `this`; see Java Bug Parade ID 4779253.

This example highlights our support for the principle of incrementality and early gratification. Only a single annotation describing a subset of the lock model—in reality the state protected by the receiver includes additional fields, and the implementation uses two other objects as locks as well—for a single class was added to the code. This single annotation is sufficient to identify a race condition; of course, the unexpressed portion of the lock model cannot be either positively or negatively assured. But the point is that the programmer only needs to annotate the portions of the model that are of current interest. The model may thus become more detailed, and its benefits accrete over time.

2.3.3. Case study: Wrong lock

Many classes maintain static fields to support unique identification of instance objects. Here is a code pattern based on code from a corporate partner:

```
1 public class C { ...
2   private static int nextID = 0;
3   private int id;
4
```

```
  protected C() { assignID(); }

  private synchronized void assignID() { id = nextID++; }
}
```

In our process of program understanding, we first attempted to model the policy using an instance lock to protect the field `id`:

```
@lock IdLock is this protects id
```

We inferred this model from the lock used in the code above. The tool informed us that none of the many other uses of the field `id` were protected by this lock. We concluded that (1) the actual state protected by `IdLock` is the `static` field `nextID`, and (2) that this state should actually be protected by locking on the class object of `C`, syntactically identified in Java by "`C.class`", rather than on the diversity of instances.[4] The repaired code is shown below; we had to change method `assignID` into `getNextID` because a `static` method cannot assign to an instance variable.

```
/** @lock IdLock is class protects nextID */
public class C { ...
  private static int nextID = 0;
  private final int id;

  protected C() { id = getNextID() }

  private static synchronized int getNextID() { return nextID++; }
}
```

2.3.4. Case study: Util.concurrent

The `util.concurrent` library contains a set of sophisticated synchronized wrapper classes for scalar types that additionally include suites of common operations that are meant to be atomic. We investigated this library with our tool. So far, we have developed 21 models within this library, and we have been able to provide positive assurance for all but two of these. One of the two exploits subtleties of the Java Memory Model and is beyond the present capabilities of the tool. The other model is, in fact, not consistent with code due to a subtle race condition detected by our tool.

This model involves the class `SynchronizedVariable` and its primitive-type–wrapping subclasses `SynchronizedInt`, `SynchronizedLong`, etc. We expressed our model by first adding region and lock declarations to the base class `SynchronizedVariable`.

```
/**
 * @region public Value
 * @lock Lock is lock_ protects Value
 */
public class SynchronizedVariable {
  protected final Object lock_;
  ...
}
```

The new region `Value` has no state in `SynchronizedVariable`; the intent is that it apply to subregions added by the various subclasses. Each subclass declares a field `value_` of

[4] Our tool does not allow a `@lock` annotation to use an instance field or `this` as a lock for `static` data because this effectively creates multiple locks for the same data.

some particular type, for which we add the design intent that it is part of the region `Value`. For example,

```
1 public class SynchronizedChar
2 extends SynchronizedVariable implements ... {
3   /** @mapInto Value */
4   protected char value_;
5   ...
6 }
```

This gives a simple example of how our assurance approach can enforce correct locking behavior *throughout* a class hierarchy. A systematic application of our approach can thus ameliorate many of the hierarchy-related issues that have caused Java's concurrency support to be criticized, e.g., [7].

In version 1.3.2 of the library, the class `SynchronizedLong` fails to assure because there is an unprotected access to `long value_` at the end of the method `swap()`; we omit the code. In general, even simple getters require locks to be held to ensure that intermediate values being computed within *another* thread are not inappropriately returned by the getter, though there are exceptions to this rule. In the specific case of 64-bit primitive types, a simultaneous access can additionally yield a bad value because two separate 32-bit accesses are needed to retrieve the composite value. A minor code change, now part of the current version of the library, allowed us to assure consistency of the code and model. (The lock was correctly held in the code for the other 64-bit primitive type, `SynchronizedDouble`.)

2.3.5. Case study: jEdit

The `jEdit` project is an open source text editor for programmers. It contains a class `BufferListSet` whose field `files` references a `String` array. Our reverse engineering suggested that the lock policy was for access to both `files` and the array it references to be protected by the `BufferListSet` object. We expressed the model as follows:

```
1 /**
2  * @region protected FilesList
3  * @lock FilesLock is this protects FilesList
4  */
5 public abstract class BufferListSet implements SearchFileSet { ...
6   /** @return {@unique} */
7   public synchronized String[] getFiles(View view) { ... }
8
9   /**
10   * @mapInto FilesList
11   * @unshared
12   * @aggregate [] into FilesList
13   */
14  private String[] files;
15
16  public void invalidateCachedList() { files = null; }
17 }
```

These annotations create (at line 2) a region named `FilesList` that includes the field `files` (line 10). The locking policy (line 3) states that `FilesList` is protected by locking the object instance. The array *referenced* by the field `files` is additionally made part of

the region `FilesList` (line 12). In our model we note that it is uniquely referenced, that is unaliased (line 11). The `@return {@unique}` annotation (line 6) indicates that `getFiles` returns a reference to an array that has no references to it from the heap.[5]

Because method `invalidateCachedList` is not `synchronized`, verification fails. This can cause a `NullPointerException` because it could set `files` to `null` after another method has checked for the fact that the field is non-`null`. The details are in jEdit Bug 893735.

From the standpoint of evolution, it is interesting to note that in both this example and the Log4j example described above, the problems were introduced when new methods (in this case, `invalidateCachedList`) were added to a class during evolution, when developers might not have had an accurate memory of original design intent.[6]

2.3.6. Discussion

We have found that attempts to fix seemingly simple verification failures often reveal deeper issues. For example, a single negative result, when coupled with numerous related positive verification results, can reveal inconsistencies in the design intent embodied in the code. This experience is similar to that of Hovemeyer and Pugh in [19], where they report that bug pattern detectors often serve as "confusion detectors". Simple examples of such situations are when a field is only sometimes accessed from a critical section, or when one lock is used sometimes to protect one field and other times another. (There are safe—i.e., race-free—ways to use these patterns, further complicating the issue.) A more complex example comes from another production system: the object used as a lock to protect a mutable field `f` is the object referenced by `f`. Thus, the lock on the field changes with the value of the field, which was not the intent.

3. Co-evolving code and models

Assurance of the consistency between design intent and code can fail for several reasons. There may be errors in the source code—bugs in the program. Or there may be errors in the models of design intent—bugs in the model or its expression as program annotations. Consistency can be restored in the first case by correcting the code and in the second case by correcting the model. There are two additional reasons: the analysis capability is insufficiently powerful to verify consistency, and it is not possible to achieve consistency between model and code, i.e., produce a race-free program, without modifying both.

In general, when resolving the model–code inconsistencies highlighted by our tool, the programmer must decide whether to trust the model or the code. Based on our experience with our tool, we have identified four evolution scenarios corresponding to the cross product of whether the programmer a priori trusts the model or the code and whether the programmer modifies the model or the code:

[5] It may have had stack-based references from local variables, but they will cease to exist when `getFiles` returns.

[6] We learned this by examining the change logs of the classes.

Trust	Modify	Scenario
Model	Code	Assured Evolution of Code
Code	Model	Reverse-Engineered Model
Model	Model	Model-Driven Evolution
Code	Code	Code-Driven Evolution

We now discuss how our tool can assist a programmer to *reestablish* code–model consistency in each scenario. Underlying this discussion is the recognition that the reality of the process is an ongoing *co-evolution* of code and design intent, which consists of many steps of the four kinds considered here.

3.1. Assured evolution of code

In general, common implementation and maintenance tasks are captured by the scenario in which the programmer trusts the design intent and modifies the code, with the intent to maintain consistency with the model. It is during these activities that the programmer can inadvertently introduce bugs while introducing new functionality or, even worse, fixing existing bugs. When using our tool, however, assurance results would indicate whether the programmer successfully maintained consistency and, perhaps more usually, focus attention to segments of code where consistency is lost, e.g., identifying a potential data race or other bug. This is analogous to regression testing. Consider again the jEdit example introduced in Section 2.3.5 in which the addition of a new method introduced a race condition. Use of our techniques could have refocused the programmer's attention from extending functionality to compliance with the locking model.

3.2. Reverse-engineered model

When the programmer trusts the code and changes the model, he or she is essentially reverse engineering the code to evolve the model to more accurately describe the implementation reality. Addition of model information, e.g., by adding annotations to code, amounts to hypothesizing a model with which the code might be consistent. Assurance tests this hypothesis. To illustrate, consider the case where a field `f` is accessed from many `synchronized` methods. The programmer might hypothesize that the field is protected by the object itself, and thus annotate the containing class with

```
@lock FLock is this protects f
```

Analysis might indicate that a majority of the uses of `f` are protected under this model, while several of them are not. Suppose further that the negative results all occur within `private` methods that are called from `synchronized` methods. The programmer can then improve the model by annotating those private methods with `@requiresLock FLock`, declaring the intent that callers should acquire the lock.

3.3. Model-driven evolution

In this scenario, the programmer both trusts and modifies the model. Negative assurance results identify segments of code that need to be modified to establish conformance with the new model. For example, we might decide that instances of `BoundedFIFO`, introduced in Section 2.3.1, should be protected by the object referenced by a new field `lock` instead of by `this`. We would update the model by changing its `@lock` annotation to be

```
@lock BufLock is lock protects Instance
```

and by adding a new field to the class:

```
public final Object lock = new Object();
```

[7]

In fact, our first step could be to alter the annotation alone: Assurance would identify that no field named `lock` exists. We would thus be guided on how to restore consistency.

Once the new lock model with the new representation of `BufLock` is declared, our tool will identify all known call sites of the methods `put` and `get` as being inconsistent with the methods' preconditions, and will identify all uses of the class's fields within the `synchronized` method `resize` as being unprotected. We already know *what* must be done to restore consistency—because we have deliberately changed the model—but we may not have known *where* in the code to implement the new model. Here, the tool results conveniently focus our attention to exactly those `synchronized` blocks that need to be updated.

3.4. Code-driven evolution

In the fourth scenario, the programmer both trusts and modifies the code—deliberately changing the design intent embodied in the code. This scenario is distinguished from Model-Driven Evolution primarily by where the programmer's focus lies. In this scenario, it is on the code: the programmer is in the midst of an implementation task and decides that task would be more easily implemented if, for example, the locking policy were different. So the programmer modifies the policy's manifestation in the code, and updates the annotated model later. Analysis assists the programmer with making the annotated model consistent with implementation by identifying the code segments that are inconsistent with the old model, and thus implicitly identifies the segments of the old model that need to be updated. Furthermore, because the tool also identifies the segments of code that have changed, the programmer knows where to look in the code for manifestations of the new evolved model.

Consider again the case of changing the lock that protects an instance of class `BoundedFIFO`. Suppose we first changed the code by introducing the new field and modifying the `synchronized` blocks and declarations as appropriate. The negative assurance results would point to the callsites of `put` and `get` and to the implementation of `resize` and report that the `BoundedFIFO` object is not being locked per the existing annotations of design intent. Again, these assurance failures suggest which segment of the model must be modified—the lock representation—and where to look for the new representation: in the `synchronized` blocks surrounding the now-inconsistent code.

4. Refactoring concurrent programs

Refactorings are patterns for systematic restructuring of code. In many tools, automatic support for refactorings is provided as program transformations. As we noted in Section 1, automated refactoring generally requires explicit modeling of programmer design intent

[7] We make the field `public` to ensure that it is visible to the clients of the object. Were we to make the field less visible, our assurance tool would motivate us to introduce a `public` "lock getter" method into the class.

to assure soundness. This means that refactoring is risky for both programmers and tool implementors. The transformed code may be less familiar and more difficult to understand than the original code, and it may also be broken because unstated preconditions are not met. For example, subtle changes to order of computation can be introduced when field declarations with initializers are hoisted to superclasses. Another example is the extraction of new method definitions that leave behind substantive computation for actual parameter values.

For some refactorings, conservative analyses can compensate for missing intent [22]. Perhaps for this reason, the Java refactoring literature has generally focused on sequential programs. With model information and supporting analyses, however, it becomes more tractable to consider more ambitious refactoring and program transformations, including manipulation of concurrent programs.

The conventional usage model for refactorings assumes that the code being refactored is initially "good"—usually with respect to an unstated model. That is, the refactoring modifies the code, implicitly trusting that at the start the code is consistent with design intent, and results in code that remains consistent with design intent—and furthermore we assume that the refactorings do not modify the usually unstated design intent.

When models are explicit, however, the picture can be slightly different: refactorings can manipulate both code and models. We can thus consider the *simultaneous* evolution of models and code rather than the cycle of model–code resynchronization described in the previous section. In this section, we consider some problems arising from traditional refactorings when there are explicit models. We then describe an approach to refactoring concurrent programs that supports co-evolution of code and models. This makes particular sense for refactorings related to the use of concurrency, because of the attribute-specific character both of the soundness preconditions for the transformation and of the models being transformed along with the code. We provide an example of one such refactoring: Split lock.

4.1. Refactorings and models

The first step we take is to incorporate the manipulation of models into the program transformation process. Without this, there can be risks to programmers making subsequent changes to refactored code.

Extract method. This refactoring replaces a programmer-selected sequence of statements with a call to a new method whose body is made up of those statements. Consider the case of extracting a sequence of statements that are nested within a `synchronized` block. Because the newly introduced call to the newly extracted method will still be within the `synchronized` block, it is easy to see that the code will be in the same state of consistency with the locking model as it was prior to the application of the refactoring.

If subsequently any new calls are introduced to the newly extracted method definition, they must adhere to the requirement to acquire the lock prior to making the call. If the locking policy is explicit, this can be handled by adding a `@requiresLock` annotation to the newly extracted method. A program transformation operating with an explicit model could do this automatically.

An extract method transformation may also interact with non-lock-based concurrent designs. For example, the AWT enforces thread safety by requiring that certain methods, such as `paint` and `update`, be executed by the "AWT Thread" only. We can capture this design intent using "thread coloring", whereby threads are abstractly identified by colors, and segments of code are colored by the threads in which they are allowed to execute [26]. Here again, extract method needs to be made aware of the design intent so that it correctly propagates the colors to the newly extracted method. Otherwise, future changes to the code could cause the new method could be run from an incorrectly colored thread.

Convert local variable to field. This refactoring replaces a method-local variable with a new field declared in the class containing the method. The danger here is that we miss an opportunity to record design intent as the new field is created. Unlike extract method, we cannot use our analyses to catch this problem after the fact because, in general, no prior models exist for the field. This refactoring, and indeed any refactoring that introduces new fields, benefits from interaction with the developer to capture the design intent for the field. To which region of state should the field belong? Is the new field intended to be accessed from multiple threads, and if so, how is access to it synchronized? Is the field intended to refer to an unaliased object? By capturing this information up front, correct use of the field can be assured from the outset.

4.2. Refactoring concurrency

The presence of explicit models of concurrency-related design intent enables implementation of refactorings that directly affect how concurrency is managed within a program. We have identified a number of transformations, listed below, that are potentially applicable in a *generative approach to concurrency management*, in which more complex concurrency is introduced in a systematic fashion. In this approach, the programmer could begin with a class definition formulated as a simple monitor—all state is protected by the object itself and every public method is `synchronized`. Such a class has a simple concurrency policy (see below): no method is allowed to interleave with any other method. The programmer then applies refactorings to modify the extent of concurrency supported by the class, updating both the code and the models describing its locking and concurrency policies. In particular, when applying refactorings, the programmer may choose to liberalize the concurrency policy.

There are several possible definitions for the "soundness" of refactorings such as those sketched here. The strongest form of soundness is a strict requirement to preserve program behavior. A less demanding requirement is to preserve model–code consistency for identified models. There may also be cases where the purpose of the refactoring is to evolve both code and model, and so some models are respected while others are transformed.

The split lock refactoring decreases the granularity at which state is protected by moving it "down" the region hierarchy. That is, a lock used to protect a single region is replaced by multiple locks used to separately protect its subregions. This increases opportunity for concurrency, but it can be dangerous if the underlying representation invariants involve close relationships among the new finer-grained regions.

The merge locks refactoring decreases the granularity of protection by moving "up" the region hierarchy, replacing multiple shared regions with a single ancestor shared region.

The shrink critical section refactoring alters the scope of the code in a critical section. The purpose is to move the boundary of a critical section past statements that do not access shared state.

The split critical section refactoring creates additional opportunities for method interleaving by converting a single `synchronized` block into a sequence of `synchronized` blocks. Subsequent refactorings of the concurrency policy could then be used to take advantage of these new opportunities. Its dual, merge critical sections, may be used to remove interleaving opportunities and reduce lock acquisition.

The synchronize method and synchronize callsite transformations modify locking responsibility, moving the responsibility between callee and caller, respectively. These transformations affect a body of code wider than the class definition because of the necessity of identifying and updating method callsites.

The naive application of refactorings such as split lock and split critical section can have adverse effects on the atomicity of operations, and possibly result in data races being introduced into the program. We temper this possibility via the application of additional programmer-specified design intent known as *concurrency policy*. The rest of this section elaborates the concept of concurrency policy and then describes split lock in more detail.

4.2.1. Preserving atomicity: Concurrency policy

A consequence of our decision to avoid the expression of representation invariants is an increased difficulty in assuring that the atomicity of operations is preserved at all appropriate levels of granularity. For example, in the following class, our lock policy analysis assures the correctness of both methods `incrementCount1` and `incrementCount2` even though the latter can result in a data race when multiple threads simultaneously invoke `incrementCount2`—or even both `incrementCount1` and `incrementCount2`—on the same object.

```
1 /** @lock CounterLock is this protects count */
2 public class Counter {
3   private int count = 0;
4
5   public synchronized void incrementCount1() {
6     this.count += 1;
7   }
8
9   public void incrementCount2() {
10    int t;
11    synchronized (this) { t = this.count; }
12    t += 1;
13    synchronized (this) { this.count = t; }
14  }
15 }
```

While the above is a contrived example, the point is that a narrow focus on only the state that must be protected is insufficient to describe the programmer's intent with respect to the extent to which particular operations are expected to be atomic. This problem is not unique to our approach: RACEFREEJAVA exhibits the same problem, and Flanagan and Qadeer have developed a type system for atomicity to compensate [13].

Our solution to this problem is the expression of *concurrency policy*, which we briefly introduce in Section 2.1. The idea of concurrency policy is to allow the programmer to declare on a per-class basis which methods are expected to have implementations that result in safe interleavings. The details are in [15]. This process actually defines abstractly what it means to be *safe*: the distinction between a race condition and an acceptable instance of non-determinism is in the mind of the programmer. Compliance with concurrency policy is enforced additionally with lock policy. It is sometimes necessary, for example, to introduce additional locks to maintain consistency with the concurrency policy: these locks are not associated with a region of state and are termed *policy locks*.

Refactoring and concurrency policy. Because concurrency policy is part of the programmer-expressed model, refactorings are expected maintain a program's consistency with concurrency policy. But splitting a lock or critical section can introduce additional interleaving opportunities for methods. Thus, refactorings such as split lock and split critical section may need to introduce policy locks to ensure that these new opportunities cannot be exploited. Generally, these new policy locks would be manifest as additional lock acquisitions that surround the critical sections manipulated by the refactoring. But the point of split lock and split critical section, however, is to enable additional interleavings to increase potential concurrency. A *static* concurrency policy therefore renders these refactorings useless—recall, in particular, that at the start of our generative approach, no method is allowed to execute interleaved with any other method in the same class. We must thus provide refactorings that empower the programmer to modify the concurrency policy of a class. Such a refactoring, in addition to modifying the annotations that define the policy, would remove or introduce acquisitions of policy locks to enable or prevent interleavings as necessary.

Thus in our notional generative approach, the first step the programmer takes to enable additional concurrency is to refactor lock models and their critical sections to introduce additional potential points of interleaving. Then the programmer modifies the concurrency policy to specify which of the possible interleavings may be exploited. Obviously, there is a balance to be achieved in the management of concurrency policy: too much interleaving risks data races—too little risks losing the advantages of concurrency. A tool can assist by identifying all possible interleavings (or the consequences of those interleavings) based on the critical sections extant in the code. But as stated above, the determination of which interleavings are harmful must be informed by intent.

One might argue that it would be easier to enforce atomicity using explicit representation invariants. But such an approach would not free the programmer from defining what is *safe*: the programmer would have to exercise the same degree of care when choosing which representation invariants define the scope of operational atomicity as would be necessary when evaluating interleavings.

4.3. The split lock refactoring

Often a class may contain independent subsets of functionality. When this is the case, the state of the object can be partitioned into subregions based on the class's functionality. Each partition, i.e., subregion, can then be protected by a different lock, providing the potential of greater concurrency among users of the object. For example, an object that

maintains a list of listeners can protect the list of listeners separately from the state used for its main task. It is thus useful to be able to "split" a single lock into multiple locks, one for each partition of state.

Our Split lock refactoring depends upon our notion of hierarchical programmer-declared regions of state. Sibling regions disjointly partition the state of their parent. Our refactoring splits the lock associated with a particular region into a set of locks, one per child region. Split lock is parameterized by the `@lock` annotation "`@lock` M `is` L `protects` R" of the lock to be split, which provides a lock L with name M associated with region R. The lock annotations within the class and all of its subclasses[8] are analyzed to determine all the child regions $R_1, \ldots, R_n$ of R. For each R_i, the programmer is asked to provide a new mutex name M_i, and to identify a `final` field of the class or `this` to be used as the lock representation L_i for that region.

Let us assume, first, that there are no `@requiresLock` annotations. We start by identifying all the `synchronized` blocks that use the given lock L.[9] For each identified block

(1) Determine the child regions $\{R_j\}$ of R that are affected by the body of the block. These are located by identifying the fields that are directly read or written by the block, as well as by identifying any regions of the class that may be indirectly affected by methods that affect objects that have been aggregated into the state of the class.[10] Combining these fields/regions with knowledge of the region hierarchy tells us which of the $\{R_j\}$, if any, the block affects.
(2) Replace the `synchronized` block with a set of nested[11] `synchronized` blocks that acquire the appropriate locks $\{L_j\}$ for the regions $\{R_j\}$.

The annotations describing the locking model of the class are updated simultaneously with the above changes to the code:

- The original `@lock` annotation is removed.
- Any annotations "`@returnsLock` M" are removed. Depending on the visibility of the regions $\{R_j\}$, it may be necessary to introduce new lock getter methods that are annotated with the appropriate `@returnsLock` annotation to ensure that the locks are as visible as the regions they protect.
- For each child region R_i of R, a new lock declaration is added to the class: "`@lock` M_i `is` L_i `protects` R_i".

[8] We must be able to determine the *complete* set of subclasses—i.e., the class may be part of a framework, and we do not have access to all the subclasses introduced by clients of the framework. This problem is not unique to concurrency; we shall say no more about it herein.

[9] We consider a `synchronized` method to be a method whose body is enclosed in `synchronized (this) { ... }`.

[10] For example, if the field `List myList` is `@unshared` and the referenced object is aggregated into the region `ListState`, then invoking `new ArrayList(this.myList)` will affect the `ListState` region of the object referenced by `this` because the constructor reads from the `Instance` region of the object referenced by `myList`, which is considered to be part of the `ListState` region of `this`.

[11] We do not consider deadlock in this paper, though in this case, it is obvious that the tool should solicit from the programmer an ordering for the new locks.

Handling `@requiresLock` annotations. We now consider the case of existing "`@requiresLock` M" annotations. These are most easily dealt with by replacing all uses of M in them with $M_1, \ldots, M_n$. This is always sound, and does not interfere with existing uses of the methods. Because this affects the locks that must be acquired at call sites, it would be done before modifying any `synchronized` blocks. This approach, however, can unnecessarily constrain the future use of the methods because they may now require more locks than they actually need, e.g., the method only accesses one subregion of R. The difficulty in automatically making the annotations less restrictive is discerning the programmer's intent: the programmer might want to use less specific annotations to preserve flexibility for future uses of the method. Thus, the programmer should be supplied with an additional refactoring to enable modification of the `@requiresLock` annotation of a method, while using analysis to prevent the programmer from underspecifying the method's locking requirements.

4.4. Related work

Our notion of concurrency policy is inspired by Schwarz and Spector's synchronized shared abstract types [25]. It is also similar to the algebra of exclusion of Noble et al. [21].

We are unaware of any specific proposals for concurrency-related refactorings. Lea describes splitting locks as a design concept [20], but does not consider it as a refactoring, nor in the context of explicitly expressed design intent. There are compiler optimizations in the literature that modify the scope of critical sections. Escape analysis can remove critical sections from classes not used in a multi-threaded manner [3,4,9,27]. Aldrich et al. present a more comprehensive analysis that removes synchronization for thread-local objects, re-entrant locks, and locks that are always acquired after another lock (so-called "enclosed locks") [1].

There are also compiler optimizations that manipulate critical sections in automatically parallelized programs to reduce lock-acquisition overhead. Plevyak, Zhang, and Chien [23] expose critical sections by inlining method calls, and then expand them to enable merging of adjacent critical sections. Diniz and Rinard [10] decrease the protection granularity in automatically parallelized object-oriented programs. All objects are originally protected by their own locks, which are then "coarsened". In subsequent work [11], they use a technique for increasing the critical section size using flow-graph reachability.

5. Conclusion

By augmenting concurrent Java programs with model information representing design intent related to "mechanical" program properties, it becomes possible to use static analyses to assure consistency of code and model. In our experiments in applying our modeling and analysis tools to a variety of existing production systems, we have been able to (1) reverse engineer to identify concurrency-related model information, (2) perform analyses to assure consistency of model with code, and (3) use modeling and analysis results to identify flaws in code. This experience illustrates the pervasiveness of race conditions, and also the potential value of more systematic approaches to developing and, particularly, evolving concurrent code. These systematic approaches range from more disciplined use of models

to tool-assisted refactorings. The refactorings, while possibly changing some aspects of program meaning, are sound in the specific sense that they maintain consistency with existing design intent. It is also possible to define refactorings whose purpose is to restructure code to accommodate particular kinds of change in design intent.

Acknowledgements

This effort was 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.

We would like to thank the anonymous reviewers for their comments. We also thank Jonathan Aldrich, John Boyland, Edwin Chan, Greg Mathis, Elissa Newman, David Swasey, Dean Sutherland, and other Fluid project members for their help.

Appendix A. Fluid annotation guide

This appendix provides a brief task-based tutorial in the use of common program annotations used for expressing Fluid models related to lock-based concurrency. (Models for non-lock concurrency and other attributes are not discussed in this paper.) More detailed descriptions of the meanings of these annotations can be found in [18,15,16,6,8]. Following conventional practice, our annotations are in the form of Javadoc-style comments, and decorate class, method, and field declarations.

A basic lock model. The canonical Java model whereby an object protects itself is declared by annotating a class with

```
1 /** @lock CLock is this protects Instance */
2 public class C { ... }
```

This (1) declares a new lock named "`CLock`", (2) identifies that lock with the instances of the class, "`this`", and (3) protects all the fields in instances of the class, the region "`Instance`". In other words, the annotation declares the programmer's intent that all accesses to instance variables of objects of class `C` and its subclasses should be from within blocks `synchronized` on the instance. The lock name enables consistent reference to the lock object in other annotations.

Extending the model: Caller locking. The programmer can declare that it is the caller's responsibility to acquire a lock by annotating a method with a declaration of this responsibility:

```
1 /** @requiresLock CLock */
2 protected void M { ... }
```

Analysis assumes that lock `CLock` is held when analyzing the implementation of method `M`, but requires that the method be invoked from within a block `synchronized` on `CLock`.

Extending the model: Aggregating arrays. An array in Java is a separate object from the object that refers to the array. Protecting an array-typed field thus protects the reference to the array only. It is not sufficient to extend the protection to the elements of the array: we also need to know that the array object is not aliased. If it were, then it would be possible to concurrently access the array even though individual references to the array are protected. Much of the time, however, it is not intended that an array is aliased; in these cases, the array can be incorporated into the state of the object that references it. We call this *aggregating state*. An array is aggregated into the object that references it by adding a pair of annotations to the array-typed field:

```
1  /**
2   * @unshared
3   * @aggregate [] into Instance
4   */
5  private Object[] myArray;
```

These annotations (1) declare the programmer's intent that the field is unaliased and (2) extend the state of the referencing object to include the elements of the array. The aliasing promise is checked by an alias analysis, while the aggregation of the array object adds to the model of what program state the programmer intends `CLock` to protect.

Unsynchronized access in constructors. Constructors cannot be declared `synchronized` in Java, but our assurance requires that fields protected by a lock only be accessed when that lock is held. During object construction, however, an object is almost always accessed by a single thread only—the thread that invoked the constructor. When this is the case, we can proceed as if the locks for the object's state are already held. The programmer can declare this "single-threaded" intent by annotating the constructor:

```
1  /**
2   * @singleThreaded
3   * @borrowed this
4   */
5  public C() { ... }
```

One way that analysis can assure the single-threadedness of the constructor is to leverage an assurance that the constructor "borrows" the constructed object—that is, that it does not create an alias to it. (The `@borrowed` annotation is further described below.) In particular, because it does not create an alias, no other thread can obtain a reference to the object under construction during the constructor's execution.

Advanced models of locked state. Our models of lock policy actually associate locks with *regions*. A region is a named, hierarchical abstraction of state. All fields are regions, and thus a region is a named, extensible set of fields. A region can be wholly contained by another region, but it cannot arbitrarily overlap with another region. All instance fields are by default children of the region `Instance`. Regions are fully described in [15,16].

The general form of the lock declaration annotation is "`@lock LockName is Lock protects Region`" where `LockName` is a programmer-declared name for the lock, `Lock` is

the reference to the lock object, and `Region` is the name of a region. The reference to the lock object may be one of the following:

- `this`, meaning the object itself is acquired to protect the state.
- A field declared in the class being annotated or an ancestor of the class being annotated that is visible within the class, e.g., it is a `protected` field from an ancestor. The field must refer to an object, e.g., the field cannot be of type `int`. The field must be `final`, otherwise the lock object that it refers to could change. The field may be `static` or instance.
- `class`, meaning the unique `Class` object referenced by the `static` pseudo-field `class`. (This is the object that is locked by `static synchronized` methods.)
- A field of an "outer" class. If the class being annotated is declared inside class `Outer`, and wants to declare that the field `f` of the instance that is the container for the inner class's instance protects a region of the inner class, then the lock reference is given by `Outer.this#f`.[12]

New regions are declared by annotating a class with "`@region Region [extends Parent]`" which declares a new region `Region` as a subregion of `Parent`. If no parent region is specified, `Instance` is used.

A field is placed in a user-declared region by annotating the field declaration:

```
1  /** @mapInto Region */
2  private int f;
```

Thus, the state of an object may be partitioned into multiple abstract regions, each protected by a different lock, enabling concurrent access to different segments of the object's state.

Method effects. Regions provide an abstract way to name the state of an object. The effects of a method—the state read and written during the execution of that method—may be expressed in terms of regions. Effects are useful when determining whether code can be reordered by a refactoring, and are also necessary to support the analyses that assure `@unshared` and `@lock` annotations are consistent with the code. An unannotated method is assumed to have the annotation "`@writes Object.All`" which declares that the method could read from or write to anything in the heap. In particular,

- If only a `@writes` (respectively, `@reads`) clause is present, the `@reads` (respectively, `@writes`) clause is assumed to be empty (`nothing`).
- Write effects include read effects.
- The region `All` is a `static` region declared in class `Object` and is the parent of the region `Instance`.

[12] While our prototype tool supports declaring locks that are the fields of outer classes, it is not presently possible to assure their correct use. This is because of deficiencies in both our internal representation and with Java syntax. In particular, given a variable `v` that refers to an instance of a non-`static` inner class, there is no syntactic expression that evaluates to the "outer" object of that instance, that is, the object referenced by `o` in the expression `o. new Inner()`. We allow the declaration of such locks, even though they cannot be assured, because we have encountered them in real code and it is important to be able to document the design intent.

Assurance checks that the actual effects of the method implementation are no greater than the declared effects. There are several fine points to this: uses of `final` fields produce no effects, effects on local variables are not visible outside of a method, effects on objects created within a method are not visible outside of a method, constructors do not have to report effects on the `Instance` region of the newly constructed object, and region aggregation (described below) is taken into account.

Here is a simple "variable" class with effects annotations:

```
1 /** @region public Value */
2 public class Var {
3   /** @mapInto Value */
4   private int value;
5
6   /** @writes nothing */
7   public Value(int v) { value = v; }
8
9   /** @reads Value */
10  public int getValue() { return value; }
11
12  /** @writes Value */
13  public void setValue(int v) { value = v; }
14 }
```

Thread effects. Methods and constructors can be annotated to declare that their invocation does not cause the execution of any new threads:

```
1   /** @starts nothing */
2   public void doesNotStartThread(...) { ... }
```

An analysis assures that the method implementation (1) does not invoke the `start` method of any thread object, and (2) only invokes methods/constructors declared not to start any threads.

Effects and `@singleThreaded` constructors. Sometimes a constructor may be single threaded, but it is not possible to annotate the constructor with "`@borrowed this`" because the constructor does in fact alias the newly created object. This often occurs when a field of the new object is initialized to point to the new object itself. To avoid this problem, a `@singleThreaded` constructor can also be assured if the constructor's write effects are bounded by `@writes Instance` and the constructor `@starts nothing`. That is, a constructor is `@singleThreaded` if it only writes to the new object or to other newly created objects and does not start any threads.

More on unshared fields. Any reference-typed field, not just arrays, can be declared to be `@unshared`. Furthermore, state aggregation allows any region of the uniquely referenced object to be aggregated into a region of the referring object as long as well-formedness rules that make sure the region hierarchy is preserved are respected. In particular, in the case of arrays, `[]` is merely the name of the region used to represent the array's elements. Thus, if objects of class `C` had regions `R1` and `R2`, then they could be separately aggregated into distinct regions of class `D` as follows:

```
1 /**
2  * @region P1
3  * @region P2
4  * @region P3
```

```
 * @lock Lock1 is this protects P1
 * @lock Lock2 is L2 protects P2
 * @lock Lock3 is L3 protects P3
 */
public class D {
  private final Object L2 = new Object();
  private final Object L3 = new Object();

  /**
   * @mapInto P1
   * @unshared
   * @aggregate R1 into P2, R2 into P3
   */
  private C myC;
  ...
}
```

Here, class `D` declares three regions. It locates the field `myC` in the first region, `P1`, and aggregates the regions of the object referenced by `myC`, `R1` and `R2`, into regions `P2` and `P3`, respectively. Each region in `D` is associated with a different lock. Thus, the reference to the `C` object, `myC`, is protected by synchronizing on `this`, while access to the regions of `R1` and `R2` of the `C` object are protected by synchronizing on `L2` and `L3`, respectively, because they are the locks associated with `D`'s regions `P2` and `P3`.

Borrowed references to unshared fields. When an object is passed as a parameter to a method, an alias to that object is created. Thus, if `@unshared` annotations are strictly enforced, an `@unshared` field can never be passed as a parameter to a method, even as the receiver. But if a method is known to not create any additional aliases to the object, then an `@unshared` field may be safely passed as a parameter because it is guaranteed that the method will restore the uniqueness of the field. A parameter (including the receiver) is declared to be borrowed by annotating the method, e.g.,

```
/** @borrowed this, array */
public void copyInternalArray(Object[] array) {
  for (int i = 0; i < array.length; i++) {
    array[i] = this.myArray[i];
  }
}
```

If this method belongs to class `C`, then it may be invoked on `@unshared` references to `C` objects; it may also be passed `@unshared` references to `Object[]` arrays. Here it is easy to see that no aliases to `this` or to `array` are created, but, in general, this is a property that is easily violated, and thus it is valuable to have an analysis that can assure that it is met.

References

[1] Jonathan Aldrich, Emin Gün Sirer, Craig Chambers, Susan J. Eggers, Comprehensive synchronization elimination for Java, Science of Computer Programming 47 (2–3) (2003) 91–120.

[2] David F. Bacon, Robert E. Strom, Ashis Tarafdar, Guava: A dialect of Java without data races, in: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2000, pp. 382–400.

[3] Bruno Blanchet, Escape analysis for object-oriented languages: application to Java, in: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1999, pp. 20–34.

[4] Jeff Bogda, Urs Hölzle, Removing unnecessary synchronization in Java, in: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1999, pp. 35–46.
[5] Gregory Bollella, James Gosling, Benjamin Brosgol, James Gosling, Peter Dibble, Steve Furr, Mark Turnbull, The Real-Time Specification for Java, Addison-Wesley, 2000.
[6] John Boyland, Alias burying: Unique variables without destructive reads, Software: Practice and Experience 31 (6) (2001) 533–553.
[7] Per Brinch Hansen, Java's insecure parallelism, ACM SIGPLAN Notices 34 (4) (1999) 38–45.
[8] Edwin C. Chan, John T. Boyland, William L. Scherlis, Promises: Limited specifications for analysis and manipulation, in: Proceedings of the 20th International Conference on Software Engineering, 1998, pp. 167–176.
[9] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, Sam Midkiff, Escape analysis for Java, in: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1999, pp. 1–19.
[10] Pedro Diniz, Martin Rinard, Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs, in: Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing, 1996, pp. 285–299.
[11] Pedro Diniz, Martin Rinard, Synchronization transformations for parallel computing, in: Proceedings of the 24th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, 1997, pp. 187–200.
[12] Cormac Flanagan, Stephen N. Freund, Type-based race detection for Java, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2000, pp. 219–232.
[13] Cormac Flanagan, Shaz Qadeer, A type and effect system for atomicity, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2003, pp. 338–349.
[14] Martin Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
[15] Aaron Greenhouse, A programmer-oriented approach to safe concurrency, Ph.D. Thesis, Carnegie Mellon University, 2003.
[16] Aaron Greenhouse, John Boyland, An object-oriented effects system, in: Object-Oriented Programming—13th European Conference, 1999, pp. 205–229.
[17] Aaron Greenhouse, T.J. Halloran, William L. Scherlis, Using Eclipse to demonstrate positive static assurance of Java program concurrency design intent, in: Proceedings of the OOPSLA Workshop on eclipse Technology eXchange, 2003, pp. 101–105. http://doi.acm.org/10.1145/965660.965681.
[18] Aaron Greenhouse, William L. Scherlis, Assuring and evolving concurrent programs: Annotations and policy, in: Proceedings of the 24th International Conference on Software Engineering, 2002, pp. 453–463.
[19] David Hovemeyer, William Pugh, Finding bugs is easy, ACM SIGPLAN Notices 39 (12) (2004) 92–106.
[20] Doug Lea, Concurrent Programming in Java, 2nd edition, Addison-Wesley, 2000.
[21] James Noble, David Holmes, John Potter, Exclusion for composite objects, in: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2000, pp. 13–28.
[22] William F. Opdyke, Refactoring object-oriented frameworks, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992.
[23] John Plevyak, Xingbin Zhang, Andrew A. Chien, Obtaining sequential efficiency for concurrent object-oriented languages, in: Proceedings of the 22nd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, 1995, pp. 311–321.
[24] William L. Scherlis, Systematic change of data representation: Program manipulations and a case study, in: Programming Languages and Systems—7th European Symposium on Programming, 1998, pp. 252–266.
[25] Peter M. Schwarz, Alfred Z. Spector, Synchronizing shared abstract types, ACM Transactions on Computer Systems 2 (3) (1984) 223–250.
[26] Dean F. Sutherland, Aaron Greenhouse, William L. Scherlis, The code of many colors: Relating threads to code and shared state, in: Proceedings of the ACM SIGPLAN–SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 2002, pp. 77–83.
[27] John Whaley, Martin Rinard, Compositional pointer and escape analysis for Java programs, in: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1999, pp. 187–206.