A Programmer-Oriented Approach to Safe Concurrency

Aaron Greenhouse

May 2003

CMU-CS-03-135

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
William L. Scherlis, Chair
Thomas Gross, Co-Chair
Guy E. Blelloch
John Boyland, University of Wisconsin–Milwaukee

Effort sponsored in part through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298, 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, and in part through the Carnegie Mellon School of Computer Science Siebel Scholars Program.

Keywords: Java, concurrency, multi-threading, analysis, annotation, software engineering tools, program understanding, assurance, thread-safety

For Irachka.

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, and the policy that distinguishes desired concurrency from race conditions. Source code often does not reveal these design decisions because they rarely have purely local manifestations in the code, or because they cannot be inferred from code. Many programmers believe it is too difficult to explicate the models in ordinary practice. As a result, this design intent is usually not expressed, and it is therefore generally infeasible to assure that concurrent programs are free of race conditions.

This thesis is about a practicable approach to capturing and expressing design intent, and, through the use of annotations and composable static analyses, assuring consistency of code and intent as both evolve. We use case studies from production Java code and a prototype analysis tool to explore the costs and benefits of a new annotation-based approach for expressing design intent. Our annotations express "mechanical" properties that programmers must already be considering, such as lock–state associations, uniqueness of references, and conceptual aggregations of state. Our analyses reveal race conditions in a variety of case study samples which were drawn from library code and production open source projects.

We developed a prototype tool that embodies static analysis techniques for assuring consistency between code and models (expressed as code annotations). Our experience with the tool provides some preliminary evidence of the practicability of our approach for ordinary programmers on deadlines. The dominant design consideration for the tool was adherence to the principle of "early gratification"—some assurance can be obtained with minimal or no annotation effort, and additional increments of annotation are rewarded with additional increments of assurance.

The novel technical features of this approach include (1) regions as flexible aggregations of state that can cross object boundaries, (2) a region-based object-oriented effects system; (3) analysis to track the association of locks with regions, (4) policy descriptions for allowable method interleavings, and (5) an incremental process for inserting, validating, and exploiting annotations.

Acknowledgements

It's been a longer journey than I originally intended, but I'm finally done with my dissertation. This was not a solitary journey, and I owe thanks to the many people who gave me support along the way. Obviously, I would like to thank my advisor, Bill Scherlis, for his invaluable advice, guidance, encouragement, and enthusiasm. I'd like to thank my co-advisor, Thomas Gross, and the rest of my committee for their time and for the helpful feedback they have provided.

My research wasn't performed in a vacuum, and without the research and engineering results of the other members of the Fluid Group, this dissertation would never have been possible. Thank you, John Boyland, Edwin Chan, Tim Halloran, Elissa Newman, Dean Sutherland, and everyone else.

I'd like to thank my wife, Irene, for her unfailing support in all things. Her exceptional patience and emotional support during the past few months have been invaluable to the completion of this dissertation and preservation of my well-being.

I'd like to thank my parents, Anna and Gerald Greenhouse, who besides bringing me into the world—a big win for me—also instilled in me the appreciation for learning and education that got me into this mess to begin with.

I'd like to thank my newly acquired in-laws, Alla and Victor Sorokorensky, for their support.

I'd like to thank my only long-term office-mate, Orna Raz, for putting up with me. I hope our discussions have been as useful for your research as they have been for mine.

On a more serious note, I would like to thank the taxpayers of the United States of America. Without *their* support my work could never have been funded. I would also like to thank Siebel Systems for their support of the Siebel Scholars Program.

On a less serious note, I would like the thank the fortune cookie that I received earlier this week for its vote of encouragement: "Soon you will be sitting on top of the world." So thank you, tasty

snack treat, and know that you were consumed to support a worthwhile activity!

Finally, I should point out that examples used in this dissertation are taken from copyrighted sources:

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reasoning about concurrent programs is a challenge for both programmers and software tool designers: What data is shared? Is it accessed safely? What locks should be held when the data is accessed? A fundamental aspect of the challenge of concurrency is that the information needed to understand a concurrent program is non-local—that is, there is no single place in the code where there is an expression of the design commitments necessary to ensure safe concurrency. Because many of these design commitments are not expressed, programmers developing and evolving concurrent code may be more likely to wander from the original design, resulting in concurrency-specific errors, such as race conditions, deadlock, and failure to comply with threading conventions of library API's. To make matters worse, when errors occur, they may be hard to localize. They may appear to be manifest in portions of code that, in fact, correctly apply the discipline. Concurrent programs are thus hard to understand and reason about, hard to evolve, and hard to assure, test, and debug.

1.1 Missing Design Information

The absence of explicit concurrency-related design information means that programmers risk introducing hard-to-find concurrency-errors during maintenance and evolution because they are forced to infer or make guesses about non-local design attributes such as lock–state association, aliasing, locking responsibility, and lock order. For any given *lock*, for example, the programmer must guess what is the state it protects. Inspection of the extant critical sections defined by that lock can suggest which variables are protected by the lock. But what about the objects referred to by those variables? Is their state separately protected? Are the references unaliased, and if so is the uniquely referenced object considered to be part of the same abstraction as the referring object? What about state that is sometimes accessed in a critical section, but sometimes not: is it meant to be protected by a lock and the program is unsafe, or are the accesses from critical sections only coincidental? For any *pair* of locks, in what order should they be acquired to prevent deadlock?

For any particular segment of *state*, source code analysis can help the programmer deduce this information from the code, but it requires inspecting *all* the portions of the code that might access that state. Such comprehensive inspection is necessary, for example, because

- A thread that accesses shared data outside of a critical section may produce data races with threads that correctly access the data from within a critical section, rendering the attempts at mutual exclusion of the correctly implemented threads irrelevant.
- Object orientation distributes conceptual resource state across multiple objects. To effectively reason about thread-safety due to object-orientation, the programmer must have a good understanding of the design intent behind class and object relationships, in particular whether the state stored across many objects, of possibly different classes, is meant to be aggregated into a single abstraction.
- A resource cannot always be reasoned about in the absence of its clients because the correct usage of the resource may depend on the design goals of its client. For example, the `length`, `get`, and `remove` methods of `java.util.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.

This non-local character of lock–state design information frustrates the later recovery of programmer design intent. Comprehensiveness also goes against the practices of using libraries and component-based architectures where the intent is to simplify program understanding by introducing abstraction boundaries. In addition, it is frequently the case that the internals of libraries and components are hidden from the client programmer, making it difficult to perform comprehensive analysis, particularly when objects are passed through abstraction boundaries.

Programmers must also guess about whether a reference is meant to be aliased, particularly in object-oriented programs. For a programmer to know that a reference to an object is unaliased is highly advantageous. Programmers frequently make this assumption, for example, when building data structures from aggregations of objects because it allows the structure to be destructively altered or objects to be reused internally without affecting clients of the data structure. In a concurrent program, the encapsulation that may be obtained by uniqueness simplifies reasoning about safety because it limits the segments of code that might alter the uniquely referenced object. If we know that a variable holds a unique reference to an object then we know that that object cannot be referenced concurrently except through the encapsulation. Thus it is enough to rely on the encapsulation to protect access to the reference and the referenced object.

Unfortunately, because aliasing is a *non-local* design attribute, reasoning about whether a reference intended to be unaliased is actually unaliased is non-trivial, and the property is easy to invalidate accidently. Inspection of the code can provide clues about the intent, but, once again, *all* uses of the field must be examined, and it is difficult to determine if an alias might inadvertently be "leaked." A programmer trying to understand the code and its design during an inspection process might think that such leaks are intentional and thus assume the reference is intended to be aliased. Explicit expression of whether a variable is intended to be unaliased would aid a programmer's understanding during maintenance and evolution, and would also enable static analysis to assure that usage is consistent with intent. Because aliasing is such a fundamental design choice for data structures, analyses assuring the correct use of a variable not intended to be aliased would be fundamental for supporting additional analyses related to other design commitments, such as lock–state association.

In general, programming languages and processes do not facilitate the capture of program design information in ways amenable or appreciable by practicing programmers. Nor do they readily provide the means to assure that captured design information is consistent with source code and vice versa. As a result, design information may be lost or out of sync with the reality of the code. Missing or incorrect design information impedes explicit assurance of the safety of concurrent programs, because the properties to assure are obscured, as well as their safe evolution. For example, absent explicit knowledge of how existing state is intended to be protected, that is, associated with locks, it is unclear how to manage the addition of new state. Is the new state to be covered by existing locks, or does it require a fresh association?

1.1.1 Kinds of Software Information

Assurance and safe evolution of concurrent programs would be easier if we could *document the missing design information* so that it is readily available to the programmer. Examples include aliasing, the extent of different segments of shared state, associations between locks and state, and which locks a method assumes the caller holds. By *missing* we mean "not easily inferable from the source code." We focus on one aspect of software design information: *models* that describe intended properties of an implementation. More specifically, we distinguish between three kinds of software

(a)
```
public synchronized void enqueue( Object o ) {
  if( size < capacity ) {
    buf[tail] = o;
    if( ++tail == buf.length ) tail = 0;
    size += 1;
  }
}
```
(b) "The lock on the queue object should be acquired before accessing any of its fields"

(c) "`tail` should always point to the next free location in the array referenced by `buf`"

Figure 1.1: Examples of software information: (a) source code, (b) design intent, and (c) design rationale.

information:

1. The *source code* of the software artifact. Source code is the ground truth with respect to what the program actually does. While analyses can be used to infer design information from source code, code is not in general fully self revealing, as we shall discuss below.
2. The *design intent* behind the source code. These are the policies that an implementation should adhere to. As discussed above, design intent is rarely captured, and even less often kept consistent with code. This research is about providing a means and incentive to capture, maintain, and exploit concurrency-related design intent. Design intent formally describes models that source code should adhere to.
3. The *design rationale* behind the policies and models that guide implementation. Design rationale is the *why* behind the *what* of the design intent. We are not attempting to capture rationale.

Consider, for example, the method in Figure 1.1(a) taken from a typical Java implementation of a queue. The actuality of implementation is that the method `enqueue` executes with the lock on the queue object held, and accesses the fields `size`, `capacity`, `tail`, and `buf`, as well as indexing into the array referenced by `buf`. The source code is consistent with the design intent—*locking model*—that the lock on the queue object should be acquired before accessing any of its fields. We can document this design intent by adding the formal annotation "`@lock QLock is this protects Instance`" to the declaration of the class. This intent is based, in part, on the representation invariant that `tail` should always point to the next free location in the array referenced by `buf`. This invariant is part of the *rationale* that we *do not* attempt to capture.

Less obvious is the design intent behind the array referenced by `buf`: the array is not intended to be shared with any other objects. More specifically, `buf` holds the only reference to the array: it is considered *unshared* or unaliased, and thus the entirely separate array object is intended to be part of the queue object. It is this intent that allows the implementation to access the contents of the array without acquiring additional locks. This design intent can be formally documented by annotating the declaration of field `buf` with "`@unshared`" and "`@aggregate [] into Instance`." These formal annotations modify the aliasing model with which source code must be consistent as well as refine the model of object state used by the previously declared locking model. These aspects of design intent are also based in part on the unstated representation invariants relating the contents of the array to the values of the head and tail indices that make up the design rationale.

The boundary between what is source code and what is design intent is not distinct. It is influenced, for example, by the kinds and levels of abstraction supported by the target programming language. Consider types in the Java and C programming languages. C has poor support for encapsulation: in spite of a facade of type definitions and structural types, data is still treated as an aggregation of bytes, as evidenced by the ubiquity of `void*` and `char*` declarations. Operations are not well associated with the data structures they are intended to apply to. Elaborate conventions on the usage of header files have evolved to address this issue. Java, on the other hand, uses classes to strongly type and encapsulate the structure of representations. Class types define the legal operations for an encapsulated data object. Information that is *tacitly captured by source code artifacts* in Java must be explicitly documented in C. Thus in Java, deliberate and accidental attempts to circumvent encapsulation boundaries or invoke incorrect operations lead to compile-time errors, while in C, the programmer must use tools such as Splint [EL02] and Rational's Purify[1] to document formally encapsulation-related design intent and to check consistency between code and intent.

1.1.2 Recording Design Intent

Similarly, *concurrency is not well abstracted in Java* (or in any other language in general use). Therefore, source code is not self-revealing with regards to concurrency-related design intent. Contributing to this problem is the lack of generally accepted 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 [GMW97], class and object design, sequencing, and state transition [BRJ99]. This dissertation thus introduces formal annotations for describing models of concurrency-related design intent such as lock–state associations, state organization and aggregation, locking responsibility, and allowable execution interleavings. Program analyses assure that code and expressed design intent are consistent, im-

[1] `http://www.rational.com`

proving assurance of code safety and the programmer's ability to safely evolve concurrent code.

We recognize that producing any kind of source code annotation, including types and Java `throws` clauses, is a burden for programmers. Programmers have shown, however, a willingness to accept this burden by adopting languages that provide sufficient payback for the annotation effort. The most successful example of a program annotation scheme is types. Programmers are willing to annotate their programs with types because it helps them understand, for example, what are the appropriate values that may be stored in a variable and what are the operations available on an object. Static compile-time type checking helps the programmer use types consistently throughout the program. In addition to being rewarded with better program understanding, the programmer is also rewarded with type safety, such as the assurance that no floating point value is going to be interpreted as an object reference. Other successful annotation schemes follow this reward model, *e.g.*, Hungarian notation [SH91], Eiffel's design-by-contract [Mey97], Java's `final` modifier, and Java's `throws` clause.

Our annotation and analysis scheme is design to offer similar incentives. We expect the burden of introducing an annotation to be low—that is, any particular annotation is relatively easy to understand and conceive. Each annotation is meant to provide immediate value to a programmer by answering a question the programmer might have about the code. A dominant design consideration for our approach is the principle of "early gratification"—some assurance can be obtained with minimal or no annotation effort, and additional increments of annotation are rewarded with additional increments of assurance. Guided by these considerations, we have developed a prototype tool that uses static analyses to assure that source code is consistent with the models described by our annotations.

1.1.3 Intent *vs*. Accident

Like type information, documented design intent is an exploitable *contract* in the program's API. Once expressed, it is thus incumbent on the source code to be consistent with the intent. As part of the API, design intent is meant to change more slowly than implementation, and hence can be engineered against in other program components. Source code, as mentioned already, reflects the truth with respect to what the program does. It is sometimes possible to infer design intent from code. For example, locks typically protect something, so analysis can be used to guess the association between locks and state. Heuristics based on common programming patterns could be used to infer relationships between objects. Even brute force can be used to exhaustively try all possible annotations [FF01]. In this work, however, we deemphasize annotation inference. From our point of view, inference has two problems: (1) it is not always possible, and (2) it is not necessarily desirable.

The expense of analysis and the unavailability of source code are among the reasons why annotations might not be inferable. Most interesting program properties can only be conservatively approximated by analysis and thus we cannot rely exclusively on inference for deriving the respective intent. Many of the properties we are interested in are, as previously discussed, non-local, that is, manifest across multiple software components and abstractions. Expensive in time and space whole-program analyses are needed to infer information in such situations. In addition, code may not be analyzable because it is part of a library distribution, because it is "native," *i.e.*, written in a language other than Java, such as Java AWT peer objects, or because it has not yet been written. Thus, explicit expression of design intent is still required to document intent at API boundaries.

More important is resolving *what is intent* from *what is accident*. That is, just because a field happens to be unaliased in the implementation, does not mean that the field is guaranteed to be unaliased. Inferred models will necessarily reflect the reality of the source code, but code does not necessarily reflect the actual design intent. Not all inferred models necessarily correspond to programmer design intent, hence the use of the word *intent*. Documenting implementation accident as design intent unnecessarily limits program evolution because of the contractual nature of design intent. Expressing more intent than is necessary thus creates more implementation obligations, and possibly—depending on the nature of the design intent—limits implementation flexibility. The benefits of expressing design intent, however, are that clients are able to exploit the stated behavior. In an earlier example, we saw how the knowledge that the array referenced by the `buf` field is unaliased simplified the overall protection of the queue's shared state.

Inference of formal annotations from source code, if performed at all, must thus be supervised by the programmer, who is responsible for determining what is design intent and what is implementation accident.

1.1.4 Assuring Consistency Between Intent and Code

Once intent is documented, maintenance and evolution tasks can be made easier because static analysis can be used to prevent the introduction of bugs into production code by *assuring* that source code is consistent with documented design intent. Thus code–design intent consistency can be preserved during maintenance and evolution which becomes focused on the aggregate of code and representations of models of design intent.

Analyses will fail to assure the annotated program when stated intent and source code are inconsistent, that is, when the stated intent fails to accurately reflect the actuality of the code. An assurance failure may be due to the *source code* being incorrect and unsafe, in which case consistency can be restored by the programmer by correcting the code. Assurance may also fail because the *stated intent* describes an incorrect model of the intended program behavior, in which case the programmer needs to correct the model describing the intent. In other words, the *assurance* provided is of consistency of the code with the models of intent.

In short, concurrent programming is difficult because design information describing non-local properties of the concurrent program is missing and it cannot be recovered from the code itself. We believe that tools to assist programmers with assuring and evolving concurrent programs must be informed by this missing concurrency-related design intent and be able to maintain the consistency between stated intent and source code as *both* evolve.

1.2 Concurrency-Related Design Intent

Maintaining consistency between stated intent and source code is a fundamental goal of this work—otherwise capturing design intent in source code is no more useful than the paper design documents that age on the programmer's shelf. More specifically, this dissertation is about (1) the nature of concurrency-related design knowledge; (2) the consequent failures of code safety that can result from the failure to express the missing design information; (3) a practicable approach to capturing and expressing the design knowledge; and (4) techniques for providing assurance that code is consistent with the captured intent as they both evolve.

The technical approach is based on source code annotation and static program analysis. A critical success factor is the feasibility and adoptability of the approach for practicing software engineers. A working programmer, who is always on a deadline should want to introduce annotations because the benefits of annotation are both useful and nearly immediate. We therefore ask the programmer to record design intent in terms of properties the programmer is already concerned about. As an example of what are these properties, and as anecdotal evidence that programmers are interested in them, we offer this excerpt from an article aimed at practicing Java programmers:

> One of the most important behaviors you should document—and which almost never is—is thread safety. Is this class thread-safe? If not, can it be made thread-safe by wrapping calls with synchronization? Must those synchronizations be relative to a specific monitor, or will any monitor used consistently be good enough? Do methods acquire locks on objects that are

visible from outside the class?[2]

Specifically, we use annotations

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

1.2.1 Establishing Safe Concurrency

We can interpret *code safety* loosely to mean that, when APIs are used as intended, “nothing bad happens.” Concurrent programming errors lead to erroneous runtime behavior that is nondeterministic and hence difficult to repeat and localize. Discovery and prevention of these programming errors and their runtime manifestations are not well covered by existing testing, debugging, and assurance techniques. We believe this is because concurrent programming errors are difficult to identify without design knowledge regarding the locking model. Current practice does not provide the incentive, the models, or a language for programmers to record this information.

Current recommended best practice for safe concurrency is to avoid using it [Lea00]. Non-expert programmers are advised to write only sequential programs. Even experts are often advised to use concurrency sparingly because it is too hard to produce large and correct concurrent programs. Consequently, less problematic alternatives to concurrency such as event-based callbacks have been advocated by researchers [Ous96]. Non-expert programmers can produce safe concurrent programs by rigorously adhering to programming patterns, such as those described in [Bir91, Lea00, Blo01a]. By following well established programming patterns known to guarantee thread safety, the programmer may feel freed from having to design the locking scheme for the program. But the programmer must still (1) implement the pattern correctly, and (2) make decisions about *which patterns* should be followed. Which objects should be aggregated into other objects, which objects should be im-

[2] Brian Goetz, “I have to document THAT?” `http://www.ibm.com/developerworks/java/library/j-jtp0821.html`.

mutable, and which objects should be shared by threads, for example. Pattern-based approaches break down when the programmer needs to do something not well covered by an existing pattern. They also have the problem that they trade assurance of thread safety for assurance of pattern compliance, and for assurance of the safety of the pattern itself.

Testing and debugging techniques that work for sequential programs do not work well for concurrent programs because of the nondeterminism in how threads may interleave their executions [MH89]. Any testing approach that attempts to cover all possible interleavings quickly becomes intractable. Testing, however, is with respect to some specification, embodied in the test cases, and thus requires knowledge of design intent. Concurrency errors are difficult to detect using code inspection because of their non-local character. But once again, for code inspection to assure the safety of the code, the programmer design intent must be known. Source level-debuggers face an additional challenge: reliably replaying a race condition once it has been discovered. This is typically done by recording event histories. Unfortunately, such approaches introduce additional nondeterminism into the program being debugged because they cause accesses to shared state that are not present in the original program. This problem has been identified as the "Heisenberg uncertainty principle" [LP85] or the"probe effect" [Gai85]. Applying model checkers to concurrent programs can be effective for discovering liveness errors, *e.g.*, deadlock, but programs must first be abstracted to produce tractable models [CDH$^+$00]. The static analyses necessary for reducing the size of the models benefit from knowledge of concurrency-related design intent, such as lock–state associations and state aggregation, although this intent is conservatively inferred in practice [Cor00]. In summary, traditional debugging techniques (1) are generally foiled by the nondeterminism and non-locality inherent in concurrent programs, and (2) typically require the knowledge of concurrency-related (and other) design intent to be fully effective.

1.3 Example: Missing Models

Let us consider as an example the models of design intent used by a collection of Java classes from the Jakarta Log4j logging library [Apa]. This library facilitates debugging by providing a framework for components to write messages of various priorities to any number of configurable logging abstractions, *e.g.*, files, consoles, e-mail. Implementations of the `Appender` interface handle the specific logging details. Appenders may be "chained" together, that is, an appender may feed into another appender to achieve additional functionality. A sophisticated logging setup could adversely affect the performance of the program: the appender may have to wait for network or disk resources that are not otherwise required by the program being examined, for example.

A multi-threaded asynchronous appender implementation `AsyncAppender` shields the program from logger latencies as much as possible—indeed, concurrency is often used to hide and manage latency is systems programs. When placed at the head of a chain of appenders, this appender decouples the logging call from the logging action by insuring that the logging activity executes in a thread distinct from the thread executing the logging request. An `AsyncAppender` simply enqueues a logging event into a buffer and returns, allowing the program to proceed with minimal interruption. The appender shares the event buffer with a distinguished dispatcher thread. This dispatcher thread removes events from the buffer when they are available and passes them to the next appender in the chain.

In this example, we concern ourselves primarily with the buffer shared between the `Async-Appender` and its dispatcher thread. In the following, we examine the models that describe the state, the aliasing relationships, and the locking conventions necessary to access the buffer in a thread-safe manner. We also describe an evolution scenario of the class and its relationship to the design models. This example illustrates (1) the non-local nature of concurrent design knowledge; the difficulties of (2) recovering the design knowledge and (3) knowing whether the intent has been correctly understood when the models are not explicit; and (4) how misunderstanding can compromise the safety of a class when it is evolved. In particular, these problems arise even when designing and using a seemingly small and simple class.

1.3.1 Class **`BoundedFIFO`**

The class `BoundedFIFO`, shown in Figure 1.2, taken from version 1.0.4 of Log4j, implements the buffer shared between `AsyncAppender` and its dispatcher thread. An instance of the class is a circular queue represented by an array referenced by `buf` and head and tail indices `first` and `next`. An instance of `BoundedFIFO` is shared by two objects, the appender object and the dispatcher thread object. A buffer instance may be accessed by code executing in at least two threads: the dispatcher thread and any thread from the rest of the program that might generate logging events—appenders may be referenced by many objects in many threads. Synchronization among the clients of `BoundedFIFO` objects is necessary, therefore, to avoid potential race conditions.

Two concurrent executions of `put`, for example, could result in the loss of an event, because the two different events could be written to the same `buf` location. This would occur if both threads executed the statement on line 24 that reads the tail index before either thread executed the statement

```
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 1.2: Class `BoundedFIFO`.

on line 25 that increments that index. Alternatively, concurrent calls to `put` could cause an `ArrayIndexOutOfBoundsException` to be thrown:

1. Suppose initially `next == size-1`.
2. The first thread puts its new element in the array by executing the statement on line 24 and then partially executes the statement on line 25 by incrementing `next` to be equal to `size`.
3. The second thread also executes the array operation on line 24 using the new value of `next` which results in an exception because `next` is now beyond the end of the array referenced by `buf`.

1.3.2 The State of BoundedFIFO

Before we consider how clients should synchronize their access to a `BoundedFIFO` instance, we need to answer the question of what is the state of a buffer instance? The five fields of the class are obviously part of the buffer's state. Traditional practice and the races described above suggest that accesses to these fields need to be coordinated. Less obvious, perhaps, is that the array referenced by `buf` should also be considered to be part of the state of the buffer because each buffer has its own *unaliased* array. The original implementor of the class clearly had this in mind, but is unlikely to have recorded it because Java does not provide a means for expressing this kind of information, and thus the programmer may not even have been explicitly aware of the design decision. We now have two elements of the concurrency model of the `BoundedFIFO` class that we can document:

1. The field `buf` is *unshared*, that is, it refers to an unaliased object
2. The state of a buffer is all its fields *plus* the state of the uniquely referenced array.

Describing these models in the code allows our tool to assure the source code of `BoundedFIFO` is consistent with the programmer's intent. If the implementation is changed so that the array is allowed to be aliased then assurance will fail. This assurance is critical to assuring that the buffer class is thread-safe. If the array is allowed to be aliased, the techniques used by clients to synchronize access to the buffer object must evolve.

1.3.3 Protecting BoundedFIFO

Now that we have a model of the state of a `BoundedFIFO` object we can consider how to synchronize access to it. The existing Log4j clients of `BoundedFIFO` are implemented to use lock-based synchronization to prevent such race conditions. Figure 1.3 reorganizes code from several

```
public class PutterClient { ...
  private final BoundedFIFO fifo;
  ...
  public PutterClient(BoundedFIFO bf, ...) { fifo = bf; ... }
  ...
  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 class GetterClient { ...
  private final BoundedFIFO fifo;
  ...
  public GetterClient(BoundedFIFO bf, ...) { fifo = bf; ... }
  ...
  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;
    }
  }
}
```

Figure 1.3: Canonical clients of `BoundedFIFO`.

Log4j classes into canonical producer and consumer clients. If we wish to add another client or to extend the functionality of `BoundedFIFO` we need to conform to the locking model used by the existing clients. This model is currently *undocumented*, although cursory inspection of the clients suggests that they use the convention of locking the `BoundedFIFO` object before invoking any of its methods. There is no single place in the code where this information is evident: a programmer trying to understand `BoundedFIFO` is forced to deduce this information by inspecting the clients of the class.

The locking model for `BoundedFIFO` thus requires that clients of a `BoundedFIFO` instance acquire the lock on that instance before invoking any methods on the object. That is, the state of the buffer, as delineated by the model given above, is protected by the lock on the object itself, and it is the responsibility of the clients to acquire that lock before invoking any of the object's methods.

In the absence of a documented model, details of the locking convention are lost. For example, is there anything significant about using the `BoundedFIFO` object as the lock? Must the lock be acquired before invoking any method, or just some of them? Is locking to protect the FIFO instance, or is it to protect the clients from inconsistent views of the FIFO? This non-locality not

```
/** Resize the buffer to a new size. If the new size is smaller than
 *  the old size events might be lost. */
public synchronized void resize(int newSize) {
  if(newSize == size) return;
  LoggingEvent[] tmp = new LoggingEvent[newSize];
  // we should not copy beyond the buf array
  int len1 = size - first;
  // we should not copy beyond the tmp array
  len1 = min(len1, newSize);
  // er.. how much do we actually need to copy?
  // We should not copy more than the actual number of elements.
  len1 = min(len1, numElts);
  // Copy from buf starting a first, to tmp, starting at position 0, len1 elements.
  System.arraycopy(buf, first, tmp, 0, len1);
  // Are there any uncopied elements and is there still space in the new array?
  int len2 = 0;
  if((len1 < numElts) && (len1 < newSize)) {
    len2 = numElts - len1;
    len2 = min(len2, newSize - len1);
    System.arraycopy(buf, 0, tmp, len1, len2);
  }
  this.buf = tmp;
  this.size = newSize;
  this.first = 0;
  this.numElts = len1+len2;
  this.next = this.numElts;
  if(this.next == this.size) // this should never happen, but again, it just might.
    this.next = 0;
}
```

Figure 1.4: Method `resize`.

only complicates our understanding of the program, but complicates maintenance as well because all client implementations must follow the same convention to maintain the integrity of the shared `BoundedFIFO` object. If the conventions change, then all the clients need to be updated to insure safety.

1.3.4 Evolution and Misunderstood Intent

The actual evolution of the Log4j library provides an interesting example of a change to `BoundedFIFO` that could be unsafe if the locking model of the class is misunderstood. Between versions 1.0.4 and 1.1b1 of Log4j, a `resize` method, shown in Figure 1.4, was added to the class. As the name implies, this method alters the capacity of the buffer. Unlike the other methods in the class, `resize` is `synchronized`, meaning that the implementation runs inside a critical section locked on the receiver object, *i.e.*, the `BoundedFIFO` object itself. A client of a FIFO object does not have to lock the FIFO before invoking this method, and indeed updated clients of `BoundedFIFO` objects do not acquire any locks before invoking it.

The implementation of method `resize` is consistent with the locking model for the class: the lock on the object is acquired before the state of the object is accessed. But the overall locking model evolves to indicate that `resize` does not require its caller to acquire the lock on the object. The implementation is also consistent with the aliasing model: the array referenced by `buf` remains unaliased, even though it may be a *different* array object after `resize` terminates. Assurance of this consistency requires assuming that `System.arraycopy` does not produce aliases to its two array parameters. This intent is recorded by annotating those two parameters as being "borrowed" which, in fact, constrains the implementation of the method.

If the locking model for the class had been misunderstood by an author of a client, then this use of the `resize` method by that client could cause a race condition. That is, supposed an author of a client misunderstood the intent, for example, by consistently, but incorrectly, using some *other* object as the lock to protect a `BoundedFIFO` instance. Now, because locking on some other object would not prevent `get` and `resize` from executing concurrently, `resize` could throw an `IndexOutOfBoundsException`. This would happen if `get` incremented `first` (Figure 1.2, line 16) after `resize` used `first` to determine the number of elements to copy (Figure 1.4, line 7) but before `resize` uses `first` as the index of where to start copying (Figure 1.4, line 14). This would cause `arraycopy` to throw an `IndexOutOfBoundsException`.

Having access to the code of `BoundedFIFO` we can deduce from this change that acquiring the lock on the `BoundedFIFO` object in clients is significant—now to do otherwise would compromise the safety of the class. If the programmer is unable to examine the source code of `BoundedFIFO` and thus unable to learn that the implementation of `resize` is `synchronized`, the situation is less clear: Is `resize` acquiring locks at all? Why do callers of the `resize` method not acquire any locks? Why must callers of the other methods acquire a lock? Is the choice of lock significant? In the absence of any explicit expression of design intent, the misunderstanding could have been on the part of the programmer who added `resize`: perhaps it was intended that clients have the freedom to use arbitrary locks—as long as all clients use the same lock—in which case `resize` should not have been made `synchronized`.

1.3.5 Summary

We have identified the following concurrency-related deign intent for `BoundedFIFO`:

- The state of a `BoundedFIFO` object must only be accessed when the object itself is locked.
- The state of a `BoundedFIFO` object includes the distinct array object referenced by `buf`; that is, the FIFO object encapsulates the array.
- We are assured that no other object has a reference to the array `buf` and thus the state of the

```
/** @lock FIFOLock is this protects Instance */
public class BoundedFIFO {
  /** @unshared
   *  @aggregate Instance into Instance */
  LoggingEvent[] buf;
  int numElts = 0, first = 0, next = 0, size;

  /** Create a new buffer of the given capacity. */
  public BoundedFIFO(int size) { ... }

  /** Returns <code>null</code> if empty.
   *  @requiresLock FIFOLock */
  public LoggingEvent get() { ... }

  /** If full, then the event is silently dropped.
   *  @requiresLock FIFOLock */
  public void put(LoggingEvent o) { ... }

  /** Get the capacity of the buffer.
   *  @requiresLock FIFOLock */
  public int getMaxSize() { ... }

  /** Get the number of elements in the buffer.
   *  @requiresLock FIFOLock */
  public int length() { ... }

  /** Returns <code>true</code> if the buffer was empty before last put operation.
   *  @requiresLock FIFOLock */
  public boolean wasEmpty() { ... }

  /** Returns <code>true</code> if the buffer was full before the last get operation.
   *  @requiresLock FIFOLock */
  public boolean wasFull() { ... }

  /** Is the buffer full?
   *  @requiresLock FIFOLock */
  public boolean isFull() { ... }

  /** Resize the buffer to a new size. If the new size is smaller than
   *  the old size events might be lost. */
  public synchronized void resize(int newSize) { ... }
}
```

Figure 1.5: Annotated class `BoundedFIFO`.

array is also protected by locking the `BoundedFIFO` object that refers to it.

- The implementation of `resize` acquires the lock appropriately, so clients do not need to acquire the lock before calling the method.
- Clients must acquire the lock before calling the other methods because they do not acquire the lock themselves. Furthermore, the clients must specifically acquire the lock that protects the state of the FIFO: the FIFO object itself.

Failure to understand and respect this intent can lead to errors during program maintenance and evolution. For example, misunderstanding the significance of the choice of lock could lead to race conditions between previously safe client code and newly added code, as described above.

1.4 Evolution and Unknown Intent

The above example illustrates some of the model elements, as well as some of the problems of recovering design intent. We now consider an example, taken from Sun's JDK, that exposes a different set of problems related to expectations about how objects are intended to be used concurrently. In particular, what methods are intended/expected to *interleave safely* when invoked concurrently, and what methods are intended to exclude each other. We call this attribute the *concurrency policy* of a class. An actual evolution scenario for the class `BufferedInputStream` of the package `java.io` provides an interesting example of the dangers of evolving an inadequately documented "thread-safe" class. This example shows another danger of inadequate documentation of design intent: when descriptions are inadequate, client programmers may develop their own design conventions based on extant component behavior, regardless of whether the behavior is intended to be a fundamental attribute of the component or not. Specifically, this example shows how insufficient documentation of design intent (1) leads to a race condition, (2) interferes with the evolution of the class, and (3) encourages clients to develop their own expectations about the concurrent behavior of the class.

The Java input–output model is stream based. Stream objects are wrapped by other stream implementations to compose functionality. A `BufferedInputStream` instance is a wrapper around another input stream object that provides buffered block-reads instead of byte-by-byte reads from the underlying stream. Documentation for the class, including the documentation for its superclass `InputStream`, is silent regarding whether the class is meant to be thread-safe. That is, is the class intended to be concurrently accessed, and if so, whose responsibility is it, caller or callee, to acquire the necessary lock(s)? Early versions of the documentation combined with inspection of the actual source code suggest that some concurrent uses are allowable and intended to be safe because some methods are `synchronized`.

The `read` method of `InputStream` is defined to block until data is available. This is a problem when using a stream backed by a network socket: if the network connection becomes hung then a `read` may block indefinitely. Java programmers developed a convention to handle this, based on the inspection process just discussed and based on observed behavior of stream implementations. The convention is to invoke `close` on the stream object *from another thread* to abort and unblock a `read` to a "stalled" socket stream. Because streams gain functionality from wrappers, this convention makes the general assumption that the methods `read` and `close` can always have interleaved executions, *i.e.*, that they do not completely exclude the execution of the other. It is also based on the assumption that closing a socket stream will affect a blocked `read`. Neither of these assumptions is grounded in behavior described in JDK documentation; they are based on empirically observed implementation-specific behavior.

In JDK 1.2, the implementation of `BufferedInputStream` was modified to prevent a race condition that could occur when the methods `read` and `close` are concurrently invoked on the same instance. The race condition occurs when `close` sets the field referring to the wrapped stream to `null` after `read` checks if the field is `null` but before `read` attempts to dereference the field. The result is a `NullPointerException` in the thread that invoked `read`. This can only happen when the stream being closed is not blocked. To prevent the occurrence of this race condition, the method `close` was made `synchronized`. Because `read` was already `synchronized`, this prevented the two methods from executing simultaneously.

This change does not appear to have been made in response to a publicly viewable bug report. A bug report related to the fix, however, was filed:[3]

> As of JDK 1.2, BufferedInputStream's close() method has had the synchronized modifier added to it. The Javadoc does not indicate that it is synchronized, but the source code does. (Actually this is pretty scary—why isn't it the same?) [...]
>
> This change in 1.2 breaks code that worked in 1.1. One technique that can be used to "free" a thread blocked on a read() is to use a second thread to close() the stream. This causes the read() to break out of its blocked state and to throw an IOException. [...]
>
> Please remove the synchronized modifier from the close() method of BufferedInputStream.

This "bug" in the fix was itself fixed by unsynchronizing `close` in the JDK 1.3 release. It remains unsynchronized in JDK 1.4. In this case, inadequate documentation of design intent (1) led to a race condition in `BufferedInputStream`; (2) caused client programmers to develop their own conventions about normative concurrent usages of the class; and (3) caused the safety of the class to be compromised for the sake of these unanticipated conventions. We note that it is possible to fix the race condition in such a way that still allows `close` to be used to unblock `read`.

This scenario occurred because of inadequate documentation of the behavior of `Buffered-InputStream` objects when accessed concurrently. The bug reporter points out that the Javadoc documentation for the class does not indicate that `close` is `synchronized`, and is disturbed that the documentation does not match the source code. It was a deliberate decision on the part of the

[3] Bug Id 4225348, "java.io.BufferedInputStream: Attempt to close while reading causes deadlock." `http://developer.java.sun.com/developer/bugParade/index.jshtml`.

Javadoc tool designers, however, to not include this information in the generated documentation:[4]

> Javadoc generates an API specification. [The keyword `synchronized` does] not belong in the signatures of a specification, because [it is] implementation-specific. ... The keyword `synchronized` indicates thread-safe behavior that should instead be described in the method descriptions. A thread-safe method itself might not use the `synchronized` keyword but might call private methods that are.

The authors of the class did not follow the above advice: `BufferedInputStream`'s method descriptions do not describe the concurrent behavior of the methods. Without this information, developers cannot understand *how* the class is thread-safe. That is, what resource–client conventions are expected to be followed to prevent the invariants of the class from being violated. In fact, they cannot know that the class is thread-safe at all.

1.5 Locking Design and Representation Invariants

Ultimately locking design conventions are rooted in design decisions about the maintenance of *representation invariants*: predicates over the state of an object that define when the object is in a self-consistent state. Operations on an object should always move the object from one consistent state to another, perhaps temporarily putting the object into inconsistent states. For example, a representation invariant for `BoundedFIFO` objects is that `next` always has a value between `0` and `size`. The `put` method might temporarily violate this invariant when it increments `next`, but then it resets `next` to `0` if it gets too big (Figure 1.2, line 25).[5] Because an operation should only be performed on an object in a consistent state, a thread must not be allowed to operate on an object if another thread may have put the object into an inconsistent state. Threads thus acquire locks when they are about to access the state of any object so that:

- They do not *operate* on an object that is in an inconsistent state. An object should always be in a consistent state when an operation begins, and the operation should return the object to a consistent state. Locking also prevents another thread from further modifying the state of an object while it is already inconsistent, which, by causing a violation of some internal precondition of the first operation, would generally make it impossible to return the object to

[4] "What's New in Javadoc 1.2.: Provides More API Information and Links." `http://java.sun.com/products/jdk/1.2/docs/tooldocs/javadoc/whatsnew.html#moreinfo`.

[5] This accounts for the more specific invariant that `next` be equal to the number of `put` operations modulo the size of the queue.

a consistent state. The race condition between `read` and `close` described in Section 1.4 is an example of this.

- They do not *view* an object that is temporarily in an inconsistent state. This is why even simple "getter" methods should be synchronized. For example, a getter might reveal an intermediate value if it could read the value of a field while that field is being used as an accumulator. Or, in the above example, another thread might view `next` when it points beyond the end of the array, before it is reset to `0`. Viewing an inconsistent state generally violates the expectations of the client.

A *race condition* is a manifestation of a failure to respect these consistency requirements. For example, concurrent `put`s cause an `ArrayIndexOutOfBounds` exception when one of the executions observes the `BoundedFIFO` object in the inconsistent state where `next` *is not less than* `size`. Concurrent executions of `get` and `resize` can interfere as described above when `get` alters `first` in such a way that it falsifies a precondition necessary for executing the `arraycopy` method on line 14: `first + len1 < size`. The point is the locking design for a class is informed by the invariants that need to be maintained for objects of the class to remain consistent. If the representation invariants were made explicit, then elements of the necessary locking design would be more clear and the rationale of the design would be more self-evident, particularly the points in the code where it is necessary to acquire locks. The designer would still have much freedom in choosing what are the particular locks associated with invariants and in delegating caller *vs.* callee locking responsibility.

Representation invariants exist at each level of abstraction. Particularly with respect to concurrency, a resource cannot always be reasoned about fully in the absence of an expression of the design expectations of its clients. For example, there is no reason why two clients cannot simultaneously invoke methods on a shared `BoundedFIFO` object, as long as the individual calls are properly synchronized. Proper locking will cause the methods to execute in *some* serialized order, but that order may not preserve the higher-level intent of the clients. A client trying to enqueue an event needs to ensure that it has a consistent view of the `BoundedFIFO`: if another client were able to invoke `put` between the time that the first client verifies that the queue is not full and the time the first client invokes `put`, the second client could cause the FIFO to be full again and the first client will have its event silently dropped. This is why the `getter` and `putter` methods in the FIFO *client* example, Figure 1.3, perform several FIFO actions within a single `synchronized` block. Understanding this, we can also hypothesize that the reason clients of `BoundedFIFO` objects are given the responsibility for acquiring the lock is that they are going to have to make a series of calls to the FIFO from within a single critical section, and thus any locking performed by the FIFO implementation would be redundant and misleading. In other words, safe implementation of a shared low-level resource with respect to its own *internal* invariants does not imply correct use of that resource with respect to the invariants of a higher-level client resource.

We believe that, in general, it is unreasonable to expect working programmers to make explicit the representation invariants of a class. It is difficult (1) to identify what are the invariants, (2) to express the invariants with sufficient formality, and (3) to assure that implementation respects the alleged invariants. Generally, a separate language of assertions is necessary to express the invariants, further raising the barrier to entry. For example, the Houdini invariant inference engine [FJL01, FL01] was developed to make the invariant documentation process less painful for users of ESC/Java [FLL$^+$02]; whether it satisfies this goal is still an open question [NE02]. Assuring consistency between source code and general representation invariants, once they are expressed, requires theorem proving. To make this process realistic for real-world programs, ESC/Java compromises both soundness and completeness. While in practice these compromises do not detract from the ability of ESC/Java to find potential programming errors, they make the approach unsuitable for providing assurance that an implementation is consistent with programmer design intent.

Our approach is to describe, instead, the mechanical attributes of the program, see Section 1.2. Although this approach is not as expressive as assertions in a general-purpose logic of programs, it does afford a means to operationalize several important attributes related to code safety and dependability. For example, our concept of *concurrency policy*—briefly introduced in Section 1.4, and described in detail in Chapter 6—is intended to replace explicit representation invariants when reasoning about how components interact in a concurrent setting. Our hypothesis is that the policy approach is considerably easier to use and is potentially more practicable than the aforementioned invariant-based approaches.

1.6 Towards A Generative Approach to Concurrency Management

The primary focus of the work described herein is the *assurance* of concurrency-related safety properties through the use of composable static analysis and design-intent–capturing program annotations. This work, however, provides the foundations for pursuing a more aggressive research goal: *establishing a principled approach to the introduction and management of concurrency, allowing the trading off of performance and concurrency to be explored without disturbing functionality and while keeping program complexity manageable*. Our goal is to provide programmers with techniques and tool support for increasing and otherwise altering concurrency during software development and evolution.

As with our assurance techniques, we hypothesize that our approach can be of value even when only minimal functional specifications are present—indeed, precisely because they cannot be counted on being present. Annotations documenting design-intent are used to assist in the safe application by a programming tool of semantics-based, meaning-preserving, source-level program transformations that alter a program's use of concurrency. A programmer requires discipline to apply a library of transformations in such a way that the annotated design intent remains respected. We have begun to define a such a discipline, termed the *generative approach*, adherence to which insures that a concurrent program is always free of the potential for race conditions, as defined by programmer-specified policy.

1.6.1 Source-level Program Transformation

Current concurrent-programming practice places much responsibility on the programmer. The programmer is responsible for

- Identifying segments of shared state that must be protected.
- Associating segments of code with the regions of state it accesses.
- Placing the appropriate locks around code segments.

Not only must the programmer do this correctly when the program is initially developed, but the programmer must also maintain the correctness of the lock placement as the program is evolved. It is perilous when the programmer wants to modify the concurrent aspects of the program.

For example, increasing the granularity of the shared segments of state, such as by using regions of objects rather than whole objects, would enable greater concurrent access to the data. Performing this modification requires identifying all the *code segments* associated with the *data regions* being partitioned, and appropriately adjusting the locking behavior of the code segments. Anecdotal evidence suggests that while this is in principle a straightforward activity, it is error prone. The programmer may miss some code segments that need to be changed, and incorrectly change other code segments, leaving shared data incorrectly protected or not protected at all. If a code segment makes use of data that is now resident in multiple data regions, and, therefore, under the protection of multiple locks, the programmer must take care to produce code that consistently acquires the locks in the same order or risk introducing potential for deadlock.

The difficulty of the tasks necessary to alter the concurrent aspects of a program can vary. Once the locks needed to protect the data accessed by a given section of code are identified, ensuring that the locks are consistently ordered is relatively easy. Identifying the regions of data accessed by a section of code is more difficult, relying on the programmer's decisions about allowable concurrency.

One of our hypotheses—to be explored separately from this dissertation—is that programmer-initiated, tool-executed, meaning-preserving, source-level program transformations can assist the programmer in "safely" evolving a program. Here, in general, by *meaning-preserving* and *safely* we mean that the transformation will not introduce race conditions or the potential for deadlock into the transformed program. One of our research issues is to develop a precise definition of *meaning-preserving*. The safety of any particular transformation is guaranteed by its preconditions which must be satisfied before it can be performed. We restrict the term *transformation* to mean a semantics-based meaning-preserving *safe* modification made to the source code. This is in contradistinction with modifications to the source code that are not safe.

1.6.2 The Generative Approach

Concurrency-related source-level program transformations by themselves are not enough to maintain the safety of a concurrent program as it evolves in a practical engineering approach. The transformations themselves must be applied with respect to a programming discipline that further insures that the annotations used by the transformations are correct and stay correct, and that manages the introduction and evolution of concurrency policy. Such a discipline we call the *generative approach*. Adherence to the generative approach insures that a concurrent program is always free of the potential for deadlock and always free of race conditions, as defined by a programmer specified policy. It is based on the observation that it is easier to stay in such a state than to initially arrive at—be proven to be in—such a state given an arbitrary concurrent program. So that the concurrent program is known to be initially safe, the generative approach prescribes a specific technique for increasing the extent of concurrency in programs: they must be generated from initial sequential programs. (Prior work, *e.g.*, [CK79, SH93, Her91], has taken this same approach; see Section 8.8.2.)

In our notional generative approach, a sequential class is made concurrent by first turning it into an approximation of a monitor [Hoa74]. This introduces an easily understood, simply implemented initial form of parallelism, together with its related annotations and an initial easily understood policy. Now that the class is "concurrent"—and importantly, known to be safe—concurrency-related transformations can be applied to the program, *e.g.*, to adjust the granularity of concurrency via a split-lock transformation, with the assurance that the program will remain safe. Our approach also suggests that the introduction of additional threads of control within a program and inter-thread signaling must also result from transformation. Because concurrent aspects of the program must be introduced and maintained via transformations, it is not possible for all safely concurrent programs to be produced using the generative approach. It is a hypothesis of this research, however, that the discipline provides sufficient flexibility to produce programs that use concurrency in a manner akin to existing real-world programs.

1.6.3 Tool Support

Transformations are performed by the tool on the behalf of the programmer and their safety is verified using program analyses. It is our belief that tool support is essential to making our approach practicable. Scaling properties of the approach are enhanced through the use of program annotations that support analyses and which provide mechanical information about the program. In particular, the annotations and analyses intended for safety assurance are reused in this role, and our regions and effects, described in Chapter 4, have their origin in our tool-supported transformation research.

Responsibility for design intent and safety is partitioned between the programmer and the tool: *the tool strictly regulates* the introduction and management of concurrency-related annotations and transformations, while *the programmer is in complete control* of the original partitioning of a class into regions and other expressions of design intent.

1.7 Outline

The remainder of this document is organized as follows. Chapter 2 provides additional background on concurrent programming in general, and on concurrency in Java, in particular. Chapter 3 describes the framework, FLUIDJAVA, we use to provide a formal description of our annotations and analyses. The following chapters focus on specific kinds of design intent: Chapter 4 describes our models of state, *regions*, and an object-oriented effects system built on top of it; Chapter 5 describes our models for associating locks with state; and Chapter 6 describes *concurrency policy*. Chapter 7 describes our prototype analysis tool and preliminary experience with it. Chapter 8 describes the *generative approach to concurrency management*, giving examples of several source code transformations and exemplifying how they might be used to evolve a concurrent program. Finally, we conclude and discuss open issues in Chapter 9.

Chapter 2

Concurrency and Java

In this chapter, we summarize (1) the programming challenges raised by concurrent programming, (2) the advice given to programmers writing concurrent programs, (3) the design intent the programmer is expected to manage, and (4) the concurrency support of the Java programming language [AGH00]. Current programming practice is, in general, still guided by recommendations that have been made since the 1960s, *e.g.*, [Dij68a, Dij68b, Hoa71, Hav68, Bir91]. Programmer adherence to these recommendations is intended to, and generally does, insure that programs are free from race conditions and deadlock. Unfortunately, the recommendations place significant responsibility on the programmer to apply the guidelines consistently and, in many cases, to maintain accurate conceptual models apart from the code itself. This reinforces the role of the annotation and assurance techniques described in the following chapters with respect to mitigating some of the programming problems.

Our work is focused on the Java programming language for three reasons. First, it is used by working programmers on production products. This makes our tools and techniques more directly adoptable, and also provides us with a large corpus of production source code from which we can obtain case studies to evaluate our work. Second, Java is a typed language with first-class encapsulation. Third, Java's concurrency features are unremarkable among modern languages, such as Modula-3[1] [Nel91], Ada 95 [TD97], and the Posix thread library [LB98]. In this regard, Java is remarkable only because it has forced concurrent programming concerns into the mainstream. Java's concurrency support is described in [GJSB00] in terms of a memory model that must be

[1]While Modula-3 has never developed a widespread programmer base, it has been influential within the research community. It is mentioned here because it is populated with "simple, safe, proven features" rather than "untried ideas" and thus it is representative of best practices in systems programming.

respected by the Java Virtual Machine (JVM), known as the Java Memory Model (JMM). This model is mostly irrelevant for correctly written concurrent programs, although several exceptions are noted below. Java's concurrency API centers around two classes, `java.lang.Thread` and `java.lang.Object`, and is described, along with general concurrent programming techniques, in many third-party books, including [Har98, Hyd99, LB99, OW99, Lea00, Hol00].

The primary message of concurrent programming guidelines is that efficiency should not be emphasized over correctness: "It is much easier to start with a correct program and work on making it efficient, than to start with an efficient program and work on making it correct" [Bir91]. This dissertation demonstrates techniques for first assuring that a concurrent program is correct (with respect to programmer design intent), and then suggests how correct concurrent programs can be systematically evolved using program transformation techniques that preserve the program's correctness.

2.1 Shared-Memory Concurrent Programming

The programming model used by Java, and the one we consider herein, is shared-memory concurrent programming. In this model, multiple threads of control—hereafter, simply *threads*—execute simultaneously, accessing data in a global memory space shared among all present and future threads. Any thread may have a reference to any object, and if more than one thread references a particular object, that object is shared, as are any of the objects reachable from that object. In *concurrent programming*, the problem is to write a single program that performs many "independent" tasks simultaneously. While concurrency can be used to increase the program performance, it is often the case that the abstractions in a program conveniently map to multiple tasks.[2] Of course, it is rarely the case that threads are completely independent, and thus they typically do share common data. It is this sharing of data that makes concurrent programming difficult and error-prone.

A program begins with a single thread, whose body is defined in Java by the `static` method `main(String[])`. Thread lifetimes are dynamic: a new thread can be created and started at any point by any other thread, and a thread completes execution when its body terminates. A program terminates when all its threads have completed. In Java, additional threads are created by instantiating instances of the standard class `java.lang.Thread` or a subclass thereof. The body of a thread is defined in one of two ways: (1) by extending `Thread` and reimplementing the `run`

[2]This dissertation does not focus on how to exploit concurrency to improve a program's performance, but rather on how to do so safely.

method, or (2) by implementing the `java.lang.Runnable` interface, which defines a `run` method only, and passing an instance of the runnable to the `Thread` constructor. Invocation of the `start` method causes a thread to begin execution; a thread completes when its `run` method returns.

The Java Memory Model defines the granularity at which memory accesses are atomic. In general, only 32-bit variables are guaranteed to be read or written atomically. The programmer must, therefore, treat shared `long` and `double` variables as if they were pairs of two 32-bit variables. If convention is followed, this is not a significant issue because in most correctly written concurrent programs *all* accesses to a shared variable—no matter its size or structure—should be from within a critical section.

2.2 Lock-Based Concurrency Management

Mutual exclusion is a design approach that limits access to shared data to a single thread at a time. Asynchrony is always resolved to a linear sequence of accesses to the shared state. That is, implementing mutual exclusion requires assuring that only one thread can access an aggregation of shared data at any given time. Other threads wait until the resource is available prior to gaining access. This assures that data invariants are intact prior to any access or mutation operation. This exclusivity of access, or "mutual exclusion," is commonly achieved by using locks, also known as mutexes, that allow threads to coordinate their actions.

A lock supports two operations: acquire and release. A thread may acquire a lock only if no other thread has already acquired it. If a thread cannot acquire a lock because another thread is holding it, then it pauses execution until some point in the future when the lock is free. When a thread releases a lock, exactly one of the threads waiting to acquire the lock is granted the lock. In Java, every object may be used as a lock, and the acquire and release operations are abstracted into a single block-structured statement: `synchronized(`*expr*`) { ... }`. The expression *expr* is evaluated first; it is a compile-time error if it is not a Java reference type, and it is a runtime error if it evaluates to `null`. Entrance into the block implicitly executes the acquire operation on the lock object, and exit from the block implicitly executes the release operation. The keyword `synchronized` may also be used as a modifier to a method definition. This is syntactic sugar for enclosing the entire method body within a `synchronized` block. For a non-`static` method, the receiver `this` is used as the lock. For a `static` method in class *C*, the `Class` object for *C* is used as the lock; there cannot be any ambiguity in the locking because `static` methods cannot be overridden. The expression `C.class` evaluates to the `Class` object for a class `C`.

Entering and leaving a `synchronized` block additionally has effects on the management of memory values in the JMM. The practical outcome of these effects is that the only way to be sure that the most recently written value of a shared variable is read is to (1) always read from that variable within a `synchronized` block and (2) always write to that value within a `synchronized` block. A correctly written concurrent program should already be following these guidelines, so this feature of the JMM should not have to be of concern to most Java programmers. The JMM insures that reads from fields declared to be `volatile` always return the most recently written value. Such fields are useful in exceptional cases, see for example Section 7.6.5, but are insufficient for maintaining representation invariants spanning multiple variables.

2.2.1 Missing Intent

In Java, as in all commonly used programming languages, it is the programmer's responsibility to produce mutual exclusion. Every shared data item—global variable, field of an object, *etc*.—should be associated with a lock, and that lock should always be used to ensure mutually exclusive access to its associated data items. It is up to the *programmer* to choose the granularity of the locking: the number of locks to use and how much, and which state is identified with any given lock. In practice, this state corresponds to the segments of state that are associated with particular data invariants. In Java, locks are most commonly associated with whole objects, although often the scope of a lock includes the state of a referenced object, consider `BoundedFIFO` in Section 1.3. It is also not unusual to protect whole sets of objects with a single lock, see Section 7.6.2. The programmer also has responsibility to ensure that the proper *sections of code* are turned into critical sections using the appropriate locks—the number and scope of the critical sections depend on the lock granularity—so that the program invariants are preserved. Critical sections are only mutually exclusive with other critical sections making use of the same lock; that is, inconsistent locking is just as bad as no locking. If the programmer accesses some shared state while not in a critical section, that access can be in a race with any of the correct critical sections that access that same state, even though those critical sections cannot otherwise race with each other. Even an access to a single scalar variable must be protected by a mutex because it too can be in an inconsistent state. Consider a variable holding a sum: if that variable is used to hold the running value as the sum is calculated, then the variable's value will be inconsistent until the calculation is complete. Unfortunately, it is common to see unsynchronized "getter" methods in Java code, even in source code from Sun's JDK [Rou03].

In other words, the programming language does not make locking mandatory within concurrent programs. Even worse, the programmer is expected to maintain a separate model of the association between a particular lock and the portions of state it is meant to protect, and then to identify the segments of code that access that state. This association and *the extent of the state* are poorly documented, easily lost, and hard to recover. One of the contributions of this dissertation is an annotation-based, tool-assisted process for capturing programmer design intent regarding the association between locks and regions of state and assuring that source code is consistent with the intent.

2.3 Condition Variables

A thread may sometimes have to wait for another thread to make true a predicate over the shared state before proceeding. The classic example is a shared queue: an item cannot be removed from the queue if the queue is empty, and an item cannot be placed in the queue if the queue is full. Thus the dequeue and enqueue operations must wait for the queue to be non-empty and non-full, respectively, before they proceed. *Condition variables* are used for this purpose. A thread invokes the wait operation on a condition variable when it wishes to wait for a predicate to become true. The waiting thread is removed from scheduling. Another thread invokes the notify operation to alert any waiters that the condition has become true. A condition variable must always be associated with a lock that is used to prevent race conditions in the condition variable itself. This lock, but not any other locks held by the current thread, is released when a thread waits, and is reacquired when the wait operation returns. In Java, each object is also a condition variable. This is supported by the `wait`, `notify`, and `notifyAll` methods of `Object`. The object uses itself as its lock, and thus it is a runtime error to invoke any of the above methods without first acquiring the lock on the object.

The guarantees provided to the awakened thread vary among implementations of condition variables [How76b]. Java does not guarantee that the condition is still true when a thread resumes, and thus the code template

```
synchronized(mutex} { ...
  while(!condition) { mutex.wait(); } ...
}
```

should be used to wait for the satisfaction of a condition. The template also demonstrates the fact that a condition variable is only *implicitly* associated with a predicate over the shared state. Again, it is the programmer's responsibility to remember these associations. The capture of design intent regarding condition variables is beyond the scope of this dissertation.

2.4 Monitors

The disciplined use of locks and condition variables to encapsulate access to an implementation of an abstract data structure has come to be known as a *monitor* [Dis71, Hoa74, BH74, BH75]. A surprising degree of variation in the behavior of monitors has developed, mostly due to how condition variables are implemented [How76a, How76b, BFC95]. Java's mutual exclusion and communication facilities were designed to make it easy to write monitors: simply declare all the methods of a class to be `synchronized` and use the receiver `this` as the one and only condition variable. If the class has non-`final` non-`private` fields, however, this is not enough to insure that the class is used in a thread-safe manner: non-`private` fields can be accessed from contexts other than the body of the class's methods, and can, therefore, be accessed without the use of synchronization. Without adhering to a strict discipline, it is, therefore, easy to write a class that may not be thread-safe. This has caused the language to be criticized [BH99b]; the proposed solution, however, requires the programmer to use only highly restrictive forms of monitors, eliminating the potential to implement classes that support high degree of concurrency, and also severely limiting the usefulness of inheritance in a concurrent setting. One of the goals of this work is to provide a framework for managing the assurance of state outside such strict encapsulations.

2.5 Additional Risks of Concurrency

The use of mutual exclusion introduces the possibility of *deadlock*. Because threads using locks must temporarily pause execution while trying to acquire a lock that is held by another thread, it is possible for a set of threads to become permanently stalled if they have a cyclic dependency on each other. Thus, the technique for avoiding one kind of error—race conditions—has the potential for introducing another kind of error. Programmers are recommended to assign a partial order to the locks used by a program; locks should always be acquired in an order that respects that partial order. Similarly, the "components" of a program should be organized in a hierarchical manner [BH74]. It should not be the case that a process placed in wait by a lower-level component can only be awakened by an action of a higher-level component. In general, this can be avoided by holding no mutex when "calling down" through the levels of abstraction, but it is often non-trivial to arrange the code so that this is the case. This avoids the *nested monitor problem* [Lis77].

Techniques for capturing design intent that may be used to assure freedom of deadlock are out of the scope of this dissertation, although the issue is briefly visited in Section 9.2.1.

Chapter 3

Recording Design Intent

Our approach to expressing models of design intent for concurrency and related design issues is to annotate Java program source code. Annotations are embedded within Java comments. This avoids any need to alter existing language tools such as compilers and IDEs. This is common practice and is precedented by such tools as LCLint [EGHT94] (now Splint [EL02]), Extended Static Checking (ESC) [DLNS98, FLL$^+$02], and Anna [Luc90]. The Java community already accepts the use of formalized comments via the Javadoc documentation generation standard. In fact, the majority of our design-intent–capturing annotations take the form of Javadoc tags. This has several advantages:

- Java programmers are already used to encountering and producing Javadoc comments.
- The Javadoc documentation engine is extensible and can be made to incorporate design rationale based on our annotations into the generated document. (We have not yet done this for our annotations.)
- Libraries exist for manipulating Javadoc within source code, simplifying our implementation effort.

3.1 Analysis and Assurance of Design Intent

Static analysis is used to assure that source code is consistent with annotated design intent. When there is an inconsistency, then either the code, the annotated design intent, or both may be wrong. In particular, design intent may be incorrectly captured or misunderstood. In general, the process for assuring consistency between annotated design intent and implementation is iterative.

Static analyses must necessarily be grounded in the semantics of the target language, Java, if assurances about program behavior are be meaningful. There is no formal semantics for the complete Java language. It is standard practice to use scaled-down versions of the Java language when describing analyses to avoid having to address features of the language that are irrelevant to the properties of interest. We thus introduce our own scaled-down Java language, FLUIDJAVA, over which we formalize our analyses. Our analyses primarily resemble type systems, and are presented as such in our mini language. We say more about the soundness of particular analyses as they are presented in later chapters.

In the following chapters we introduce annotations for recording concurrency-related design intent, extend our mini Java language to support those annotations, and describe analyses for providing assurance based on those annotations. For clarity, source code examples are always in full Java with annotations; use of FLUIDJAVA language is generally reserved for formal presentations of analysis.

3.2 The FLUIDJAVA Language

We use a mini-language called FLUIDJAVA[1] based on the CLASSICJAVA and CONCURRENTJAVA languages of [FKF98] and [FF00, BR01, BLR02], respectively. CLASSICJAVA was originally used as the base language for studying models of inheritance in Java. Features it is missing include scalar types, mutable local variables, and concurrency. As a small Java-like language, however, it has also been used by others, *e.g.*, [FF00, BR01, Yat99], as the basis for presenting analyses for Java programs. In particular, [FF00, BR01, BLR02] derive a mini-language supporting concurrency from CLASSICJAVA that they call CONCURRENTJAVA.

FLUIDJAVA is basically CLASSICJAVA with the concurrency features, `final` variables, mutable local variables, and `int` and `boolean` scalar types of CONCURRENTJAVA, but with interfaces removed from the language. Additional differences are noted as relevant. Mutable local variables, `final` variables, and scalar types are interesting for effects analysis.

3.2.1 The Language

The grammar for FLUIDJAVA is shown in Figure 3.1. A program is a sequence of class definitions followed by an initial expression that replaces the standard Java `public static void main` method. The class `Object` is predefined and is the root of the class hierarchy. Programmer-defined

[1]This research was performed as part of the Fluid Project, hence the name.

$$
\begin{array}{rcl}
P & ::= & \mathit{defn}^*\ e \\
\mathit{defn} & ::= & \textsf{class}\ cn\ \textsf{extends}\ c\ \{\mathit{field}^*\ \mathit{meth}^*\} \\
\mathit{field} & ::= & \mathit{mod\_t}\ \mathit{fd} = e \\
\mathit{meth} & ::= & t\ mn(\mathit{arg}^*)\ \{\mathit{body}\} \\
\mathit{body} & ::= & e \mid \textsf{abstract} \\
\mathit{arg} & ::= & \mathit{mod\_t}\ x \\
\mathit{mod\_t} & ::= & [\textsf{final}]_{\mathrm{opt}}\ t \\
t & ::= & c \mid \textsf{int} \mid \textsf{boolean} \\
c & ::= & cn \mid \textsf{Object} \\
e & ::= & \textsf{null} \mid \textsf{new}\ c \mid x \mid x = e \mid e.\mathit{fd} \mid e.\mathit{fd} = e \mid e.mn(e^*) \mid \textsf{super}.mn(e^*) \\
& \mid & e_1 ; e_2 \mid \textsf{let}\ \mathit{mod\_t}\ x = e\ \textsf{in}\ \{e\} \mid \textsf{if}(e)\ \{e\}\ \textsf{else}\ \{e\} \mid \textsf{synchronized}(e)\ \{e\} \mid \textsf{fork}\ \{e\}
\end{array}
$$

Identifiers are drawn from the following distinct name spaces:
$cn \in$ class names $\mathit{fd} \in$ field names $mn \in$ method names $x \in$ variable names

Figure 3.1: The grammar for FLUIDJAVA.

classes must extend exactly one class; this guarantees that all classes are comparable with `Object`. Variable and field declarations always include initialization expressions and may optionally be declared as `final`. Additional points to note about the language are:

- The receiver this is not treated as a distinguished expression, but is instead a final local variable added to the environment when type checking method bodies.
- Fields and methods do not have visibility modifiers: they are all effectively `public`.
- The name spaces for methods, fields, classes, and local variables are distinct and disjoint.
- Methods are declared to be `abstract` via the method body {abstract}.
- Classes do not have constructors. Instead all fields have explicit initializers.
- While CLASSICJAVA allows fields to be shadowed, we have chosen to remove this feature in order to simplify reasoning about fields, and in later chapters, regions.
- Unlike the full Java language in which threads are special objects, in FLUIDJAVA, threads are created using an explicit fork expression that executes its body in a newly created thread. The result of this expression is uninteresting and is typed as an int. Traditional Java `Thread` objects can be translated straightforwardly; Figure 3.2 shows one possible translation.

3.2.2 Language Predicates

Several predicates and relations are used to reason over a program P. These are based on the predicates from [FKF98] and are shown in Figure 3.3. These predicates are used to insure the program

Java	FLUIDJAVA

<pre>public class MyThread extends Thread {
  private int x = 0;
 	
  public void run() { ... }
 	
  public static void
  main(String args[]) {
    MyThread t = new MyThread();
    t.start();
  }
}</pre>
	<pre>class MyThread extends Object {
	  int x = 0
	
	  int run() { ... }
	}
	
	let MyThread t = new MyThread
	in { fork { t.run() } }</pre>

Figure 3.2: One possible translation of a Java thread class into FLUIDJAVA.

$\textbf{ClassOnce}(P) \equiv \forall c, c' : \textsf{class}\ c \cdots \textsf{class}\ c' \cdots \text{ is in } P \Rightarrow c \neq c'$
$\textbf{FieldsOnce}(P) \equiv \forall fd, fd' : \textsf{class} \cdots \{\cdots t\ fd \cdots t'\ fd' \cdots\} \text{ is in } P \Rightarrow fd \neq fd'$
$\textbf{MethodsOnce}(P) \equiv$ $\forall mn, mn' : \textsf{class} \cdots \{\cdots mn(\cdots)\{\cdots\} \cdots mn'(\cdots)\{\cdots\} \cdots\} \text{ is in } P \Rightarrow mn \neq mn'$
$c \prec^{c} c' \Leftrightarrow \textsf{class}\ c\ \textsf{extends}\ c' \cdots \{\cdots\} \text{ is in } P$
$\langle c.fd, mod\_t \rangle \mathrel{\in\!\!\in}^{c} c \Leftrightarrow \textsf{class}\ c \cdots \{\cdots mod\_t\ fd \cdots\} \text{ is in } P$
$\langle mn, t_1 \ldots t_n \rightarrow t, (x_1 \ldots x_n), e \rangle \mathrel{\in\!\!\in}^{c} c \Leftrightarrow \textsf{class}\ c \cdots \{\cdots t\ mn([\textsf{final}]_{\mathrm{opt}}\ t_1\ x_1 \ldots [\textsf{final}]_{\mathrm{opt}}\ t_n\ x_n)\{e\} \cdots\} \text{ is in } P$
$\leq^{c} \equiv$ the transitive, reflexive closure of $\prec^{c}$
$\textbf{NoShadowing}(P) \equiv$ $\langle c.fd, mod\_t \rangle \mathrel{\in\!\!\in}^{c} c \Rightarrow (\nexists c', mod\_t' : c \neq c' \wedge c \leq^{c} c' \wedge \langle c'.fd, mod\_t' \rangle \mathrel{\in\!\!\in}^{c} c')$
$\textbf{CompleteClasses}(P) \equiv \mathrm{rng}(\prec^{c}) \subseteq \mathrm{dom}(\prec^{c}) \cup \{\textsf{Object}\}$
$\textbf{WFClasses}(P) \equiv\ \leq^{c}$ is antisymmetric
$\textbf{ClassMethodsOK}(P) \equiv$ $\forall c, c', e, e', mn, T, T', V, V' : (\langle mn, T, V, e \rangle \mathrel{\in\!\!\in}^{c} c \wedge \langle mn, T', V', e' \rangle \mathrel{\in\!\!\in}^{c} c') \Rightarrow (T = T' \vee c \not\leq^{c} c')$
$\langle c'.fd, mod\_t \rangle \in^{c} c \Leftrightarrow \langle c'.fd, mod\_t \rangle \mathrel{\in\!\!\in}^{c} c' \wedge c \leq^{c} c'$
$\langle mn, T, V, e \rangle \in^{c} c \Leftrightarrow$ $\langle mn, T, V, e \rangle \mathrel{\in\!\!\in}^{c} c' \wedge c' = \min\{c'' \mid c \leq^{c} c'' \wedge \exists e', V' : \langle mn, T, V', e' \rangle \mathrel{\in\!\!\in}^{c} c''\}$
$\textbf{NoAbstractMethods}(P, c) \equiv \forall mn, T, V, e : \langle mn, T, V, e \rangle \in^{c} c \Rightarrow e \neq \textsf{abstract}$

Figure 3.3: Predicates and relations in the model of FLUIDJAVA. Based on Figure 4 of [FKF98]. The relation "is in" relates arbitrary symbol sequences. Centered ellipses "$\cdots$" are an arbitrary sequence of symbols with balanced braces and parentheses, while "$\ldots$" indicate a repeated pattern or continued sequence. The meta-variable T abbreviates method signatures $t_1 \ldots t_n \rightarrow t$, and V is used for variable lists $(x_1 \ldots x_n)$. Relations are implicitly parameterized by the program P.

Judgment	*Meaning*
$\vdash P : t$	Program P yields type t
$P \vdash \mathit{defn}$	defn is a well formed definition
$P \vdash \mathit{mod\_t}$	Type $\mathit{mod\_t}$ exists
$P \vdash \mathit{field}$	field is a well formed field
$P, c \vdash \mathit{meth}$	meth is a well formed method in type c
$P; E \vdash e : t$	expression e has type t

Figure 3.4: Typing judgments for FLUIDJAVA.

is well formed. $\mathbf{ClassOnce}(P)$ insures that class names are declared only once. Within each class definition, $\mathbf{FieldsOnce}(P)$ and $\mathbf{MethodsOnce}(P)$ insure that field and method names are unique. Predicate $\mathbf{NoShadowing}(P)$ is satisfied when a class definition does not shadow any fields declared in its ancestors.

$\mathbf{CompleteClasses}(P)$ checks that classes that are extended are defined; $\mathbf{WFClasses}(P)$ checks that the class hierarchy is an order (no loops). $\mathbf{ClassMethodsOK}(P)$ insures that method overriding preserves type. Finally, $\mathbf{NoAbstractMethods}(P, mn)$ is used as an antecedent to the new evaluation rule to prevent the instantiation of classes with abstract methods.

The transitive–reflexive closure of the immediate subclass relation $\prec^{c}$ is used to define the subclass relation: $\leq^{c}$. The relation $\Subset^{c}$ records what fields and methods are declared in a particular class, while the relation $\in^{c}$ records what fields and methods are contained in a class.

3.2.3 Typing Rules

Typing environments keep track of local named state: they have the form

$$E \quad ::= \quad \emptyset \mid E, \mathit{mod\_t}\ x$$

The predicate $x \in E$ tests whether variable x is defined in an environment E:

$$x \in E \quad \Leftrightarrow \quad E = E_1, \mathit{mod\_t}\ x, E_2$$

Figure 3.4 shows the forms of the typing judgments for FLUIDJAVA. The basic typing rules are shown in Figure 3.5. These rules are based on those of [FKF98] without the type elaboration, but are presented in a form more similar to that of [FF00]. Here we briefly describe the typing rules.

$$\textsc{Prog}\quad \frac{\begin{array}{c}\textbf{ClassOnce}(P) \quad \textbf{FieldsOnce}(P) \quad \textbf{MethodsOnce}(P) \quad \textbf{NoShadowing}(P) \quad \textbf{CompleteClasses}(P)\\ \textbf{WFClasses}(P) \quad \textbf{ClassMethodsOK}(P) \quad P = \mathit{defn}_1 \ldots \mathit{defn}_n\ e \quad P \vdash \mathit{defn}_i \quad P; \emptyset \vdash e : t\end{array}}{\vdash P : t}$$

$$\textsc{Class}\quad \frac{P \vdash \mathit{field}_i \qquad P, cn \vdash \mathit{meth}_i}{P \vdash \mathsf{class}\ cn \cdots \{\mathit{field}_1 \ldots \mathit{field}_j\ \mathit{meth}_1 \ldots \mathit{meth}_k\}} \qquad \textsc{Field}\quad \frac{P \vdash t \qquad P; \emptyset \vdash e : t}{P \vdash [\mathsf{final}]_{\mathrm{opt}}\ t\ \mathit{fd} = e}$$

$$\textsc{Method}\quad \frac{P \vdash t \qquad P \vdash \mathit{mod\_t}_i \qquad P; \mathsf{final}\ c\ \mathsf{this}, \mathit{mod\_t}_1\ x_1, \ldots, \mathit{mod\_t}_n\ x_n \vdash e : t}{P, c \vdash t\ mn(\mathit{mod\_t}_1\ x_1 \ldots \mathit{mod\_t}_n\ x_n)\ \{e\}} \qquad \textsc{FinalType}\quad \frac{P \vdash t}{P \vdash \mathsf{final}\ t} \qquad \textsc{Int}\quad \frac{}{P \vdash \mathsf{int}}$$

$$\textsc{Bool}\quad \frac{}{P \vdash \mathsf{boolean}} \qquad \textsc{Obj}\quad \frac{t \in \mathrm{dom}(\prec^{\mathrm{c}}) \cup \{\mathsf{Object}\}}{P \vdash t} \qquad \textsc{Sub}\quad \frac{P; E \vdash e : c' \qquad c' \leq^{\mathrm{c}} c}{P; E \vdash e : c} \qquad \textsc{Null}\quad \frac{P \vdash c}{P; E \vdash \mathsf{null} : c}$$

$$\textsc{New}\quad \frac{P \vdash c \qquad \textbf{NoAbstractMethods}(P, c)}{P; E \vdash \mathsf{new}\ c : c} \qquad \textsc{Var}\quad \frac{E = E_1, [\mathsf{final}]_{\mathrm{opt}}\ t\ x, E_2}{P; E \vdash x : t} \qquad \textsc{Assign}\quad \frac{P; E \vdash e : t \qquad E = E_1, t\ x, E_2}{P; E \vdash x = e : t}$$

$$\textsc{Get}\quad \frac{P; E \vdash e : c \qquad \langle c'.\mathit{fd}, [\mathsf{final}]_{\mathrm{opt}}\ t\rangle \in^{\mathrm{c}} c}{P; E \vdash e.\mathit{fd} : t} \qquad \textsc{Set}\quad \frac{P; E \vdash e : c \qquad \langle c'.\mathit{fd}, t\rangle \in^{\mathrm{c}} c \qquad P; E \vdash e' : t}{P; E \vdash e.\mathit{fd} = e' : t}$$

$$\textsc{If}\quad \frac{P; E \vdash e_1 : \mathsf{boolean} \qquad P; E \vdash e_2 : t \qquad P; E \vdash e_3 : t}{P; E \vdash \mathsf{if}(e_1)\ \{e_2\}\ \mathsf{else}\ \{e_3\} : t}$$

$$\textsc{Let}\quad \frac{\mathit{mod\_t} = [\mathsf{final}]_{\mathrm{opt}}\ t \qquad x \notin E \qquad P; E \vdash e_1 : t \qquad P; E, \mathit{mod\_t}\ x \vdash e_2 : t'}{P; E \vdash \mathsf{let}\ \mathit{mod\_t}\ x = e_1\ \mathsf{in}\ \{e_2\} : t'} \qquad \textsc{Sync}\quad \frac{P; E \vdash e_1 : c \qquad P; E \vdash e_2 : t}{P; E \vdash \mathsf{synchronized}(e_1)\ \{e_2\} : t}$$

$$\textsc{Fork}\quad \frac{P; E \vdash e : t}{P; E \vdash \mathsf{fork}\ \{e\} : \mathsf{int}} \qquad \textsc{Invoke}\quad \frac{P; E \vdash e : c \qquad P; E \vdash e_i : t_i \qquad \langle mn, t_1 \ldots t_n \to t, V, e_b\rangle \in^{\mathrm{c}} c}{P; E \vdash e.mn(e_1 \ldots e_n) : t}$$

$$\textsc{Super}\quad \frac{P; E \vdash \mathsf{this} : c' \qquad c' \prec^{\mathrm{c}} c \qquad \langle mn, t_1 \ldots t_n \to t, V, e_b\rangle \in^{\mathrm{c}} c \qquad e_b \neq \mathsf{abstract} \qquad P; E \vdash e_i : t_i}{P; E \vdash \mathsf{super}.mn(e_1 \ldots e_n) : t}$$

$$\textsc{Seq}\quad \frac{P; E \vdash e_1 : t_1 \qquad P; E \vdash e_2 : t_2}{P; E \vdash e_1; e_2 : t_2} \qquad \textsc{Abstract}\quad \frac{P \vdash t}{P; E \vdash \mathsf{abstract} : t}$$

Figure 3.5: Typing rules for FLUIDJAVA.

A program evaluates to type t if all its class definitions are well formed, the previously described predicates are satisfied, and the body of the program evaluates to type t. A class definition is well formed if all the field and method declarations it contains are well formed. Field and method declarations are checked in an environment in which the receiver is declared to be final, preventing object identity from being changed.

A field is well formed if its declared type exists and the type of its initialization expression is compatible with the declared type of the field. Proving this may require intervening usages of the subtyping rule SUB. A field's initializer is checked in an environment where the receiver this *is not* defined. This prevents initialization expressions from referring to fields that have not yet been initialized. It also prevents them from invoking methods on the object, which is problematic because methods may also refer to fields that are not yet initialized. A method is well formed if its return type and the types of its arguments exist and the body of the method can be proven to have the declared return type. The ABSTRACT typing rule allows abstract method bodies to have any type so that abstract methods are well formed.

A type exists if it is the primitive type int or boolean, the built in reference type Object, or the name of a class declared in the program.

The subtyping rule SUB allows the type of an expression to move up the type hierarchy. A null expression can be typed to any existing class type. A new c expression has type c if c is an existing non-abstract class type; the syntactic restriction of c to the class name space prevents instantiation of non-class types.

The type read from a local variable is according to the typing environment. The type of assigning to a non-final local variable is also according to the typing environment, but only if the rvalue expression has the appropriate type; a final variable cannot be assigned to.

The type read from an object field is according to the declared type of the field in the object's type. Assignment to a non-final field has the type of the field if the rvalue expression is of the appropriate type.

The typing of the if, let, synchronized, fork, and sequencing expressions is straightforward, although the following points are worth noting: the let expression allows read-only variables to be declared and prevents the shadowing of local variable names, e_1 must type to a reference type for the synchronized expression, and the resulting value of fork expression is uninteresting and thrown away.

A method invocation has the return type of the invoked method if the receiver expression has a

$$\begin{array}{rcl} meth & ::= & t\ mn_{lbl}(arg^*)\ \{body\} \\ arg & ::= & mod\_t\ [x]_{lbl} \\ e & ::= & \ldots \mid [e]_{lbl} \end{array}$$

Labels are drawn from a new name space, distinct from the others: $lbl \in \text{expression labels}$

Figure 3.6: FLUIDJAVA grammar with labeled expressions.

PROG

$$\frac{\begin{array}{c}\textbf{ClassOnce}(P) \\ \textbf{FieldsOnce}(P) \quad \textbf{MethodsOnce}(P) \quad \textbf{NoShadowing}(P) \quad \textbf{CompleteClasses}(P) \quad \textbf{WFClasses}(P) \\ \textbf{ClassMethodsOK}(P) \quad \textbf{UniqueLabels}(P) \quad P = defn_1 \ldots defn_n\ e \quad P \vdash defn_i \quad P;\emptyset \vdash e : t\end{array}}{\vdash P : t}$$

METHOD

$$\frac{\begin{array}{c}P \vdash t \qquad P \vdash mod\_t_i \\ E = \mathsf{final}\ c\ \mathsf{this}, mod\_t_1\ x_1, \ldots, mod\_t_n\ x_n \quad P;E \vdash e : t \quad \mathcal{E}_P \supseteq \{(lbl,\ E), (lbl_1,\ E), \ldots, (lbl_n,\ E)\}\end{array}}{P, c \vdash t\ mn_{lbl}(mod\_t_1\ [x_1]_{lbl_1} \ldots mod\_t_n\ [x_n]_{lbl_n})\ \{e\}}$$

LABEL

$$\frac{P;E \vdash e : t \qquad \mathcal{E}_P \supseteq \{(lbl,\ E)\}}{P;E \vdash [e]_{lbl} : t}$$

Figure 3.7: Type rules for labeled expressions.

reference type and the arguments evaluate to the appropriate types. The class member relation $\in^{c}$ takes method overriding into account. The superclass invocation expression types similarly except that it starts the method lookup in the super class and must insure that the found declaration is non-abstract.

3.3 Labeling Expressions

We wish to be able to identify where in a program an expression comes from. This is used to make the "context" of an expression implicit, and is also fundamental to our alias resolution scheme. We thus modify the grammar of FLUIDJAVA so that expressions may be uniquely labeled. The labels differentiate otherwise lexically identical expressions based on their position in the program. We also label the formal parameters of method declarations; the method declaration itself is labeled to provide a label for the implicit parameter this. In our implementation, we use the position of an expression or declaration in the Java parse tree as the label. Labels are added to the grammar by adding a new labeled expression, and by modifying the *meth* and *arg* productions. This makes labeled expressions optional, but aliasing resolution and effects comparison as described in Section 4.11 will not work in general unless all expressions are labeled. The modifications to the grammar are shown in Figure 3.6.

For labels to be useful, each expression must be *uniquely* labeled. $\mathbf{UniqueLabels}(P)$ checks this property, specifically that every label in a program P is unique, and is added to the antecedents of the PROG rule; see Figure 3.7. The definition of the predicate is obvious, but tedious, and so is omitted. The definition of $\Subset^{c}$ for methods is updated to ignore the labels on the formal parameters (see, for example, the definition in Figure 4.7). Labels allow us to reconstruct the environment of an expression based on its label. For a well formed program P, *i.e.*, a program with unique labels, we define the function $\mathcal{E}_P$ that maps a label to an environment. This function is defined using a set constraint as an antecedent to the new LABEL rule, which otherwise propagates the type of the underlying expression; we also modify METHOD to assign an environment to the labels of the formal parameters.

Labels *do not* affect the equality of variable names when looking up variables in the environment. In general, we elide labels when appropriate.

3.4 Binding Context Analysis

We need to be able to track the mutable state references that could occur in local variables. We use what we call "Binding Context Analysis" to compute a relation between local variables and such references that each variable may be equal to. The relation also permits a local to be paired with the initial value of a formal method parameter. This analysis is similar to def–use analysis for local variables, except that it traces *through* local variable assignments to determine the ultimate source of the value that reaches a particular variable use. The set of variable bindings is the set of program expressions. The binding relations are drawn from the set $\mathbf{VB}$ for "variable bindings:"

$$\begin{aligned} \mathbf{Bindings} &= e \\ \mathbf{VB} &= 2^{(\mathsf{this} \,\cup\, \text{variable names}) \times \mathbf{Bindings}} \end{aligned}$$

The function $B\ e\ V$ gives the set of possible bindings for a labeled expression e based on the binding relation V.

$$\begin{aligned} B &: e \rightarrow \mathbf{VB} \rightarrow 2^{\mathbf{Bindings}} \\ B\ e\ V &= \begin{cases} \{b \mid (x,b) \in V\} & (e \equiv [x]_{lbl}) \\ \{\} & (e \equiv [\mathsf{null}]_{lbl}) \\ \{e\} & \text{(Otherwise)} \end{cases} \end{aligned}$$

$meth$	$::=$	$t_0\ mn_{lbl}(mod\_t_1\ [x_1]_{lbl_1} \ldots mod\_t_n\ [x_n]_{lbl_n})\{[e]_{lbl}\}$	$V^-_{lbl} = \{(\mathsf{this},\ [\mathsf{this}]_{lbl}), (x_1,\ [x_1]_{lbl_1}), \ldots, (x_n,\ [x_n]_{lbl_n})\}$
e	$::=$	$[\mathsf{null}]_{lbl}$	$V^+_{lbl} = V^-_{lbl}$
e	$::=$	$[\mathsf{new}\ c]_{lbl}$	$V^+_{lbl} = V^-_{lbl}$
e	$::=$	$[x]_{lbl}$	$V^+_{lbl} = V^-_{lbl}$
e	$::=$	$[x = [e']_{lbl'}]_{lbl}$	$V^-_{lbl'},\ V^+_{lbl} = V^-_{lbl},\ \left((V^+_{lbl'} \backslash x) \cup \{(x, b) \mid b \in B\ [e']_{lbl'}\ V^+_{lbl'}\}\right)$
e	$::=$	$[[e']_{lbl'}.fd]_{lbl}$	$V^-_{lbl'},\ V^+_{lbl} = V^-_{lbl},\ V^+_{lbl'}$
e	$::=$	$[[e_0]_{lbl_0}.mn([e_1]_{lbl_1} \ldots [e_n]_{lbl_n})]_{lbl}$	$V^-_{lbl_0}, \ldots, V^-_{lbl_n},\ V^+_{lbl} = V^-_{lbl},\ V^+_{lbl_0}, \ldots, V^+_{lbl_n}$
e	$::=$	$[\mathsf{super}.mn([e_1]_{lbl_1} \ldots [e_n]_{lbl_n})]_{lbl}$	$V^-_{lbl_1}, \ldots, V^-_{lbl_n},\ V^+_{lbl} = V^-_{lbl},\ V^+_{lbl_1}, \ldots, V^+_{lbl_n}$
e	$::=$	$[[e_1]_{lbl_1}; [e_2]_{lbl_2}]_{lbl}$	$V^-_{lbl_1},\ V^-_{lbl_2},\ V^+_{lbl} = V^-_{lbl},\ V^+_{lbl_1},\ V^+_{lbl_2}$
e	$::=$	$[\mathsf{let}\ [\mathsf{final}]_{\mathrm{opt}}\ t\ x = [e_1]_{lbl_1}\ \mathsf{in}\ \{[e_2]_{lbl_2}\}]_{lbl}$	$V^-_{lbl_1},\ V^-_{lbl_2},\ V^+_{lbl} = V^-_{lbl},\ \left((V^+_{lbl_1} \backslash x) \cup \{(x, b) \mid b \in B\ [e_1]_{lbl_1}\ V^+_{lbl_1}\}\right),\ V^+_{lbl_2}$
e	$::=$	$[\mathsf{if}([e_1]_{lbl_1})\ \{[e_2]_{lbl_2}\}\ \mathsf{else}\ \{[e_3]_{lbl_3}\}]_{lbl}$	$V^-_{lbl_1},\ V^-_{lbl_2},\ V^-_{lbl_3},\ V^+_{lbl} = V^-_{lbl},\ V^+_{lbl_1},\ V^+_{lbl_1},\ (V^+_{lbl_2} \cup V^+_{lbl_3})$
e	$::=$	$[\mathsf{synchronized}([e_1]_{lbl_1})\ \{[e_2]_{lbl_2}\}]_{lbl}$	$V^-_{lbl_1},\ V^-_{lbl_2},\ V^+_{lbl} = V^-_{lbl},\ V^+_{lbl_1},\ V^+_{lbl_2}$
e	$::=$	$[\mathsf{fork}\ \{[e']_{lbl'}\}]_{lbl}$	$V^-_{lbl'},\ V^+_{lbl} = V^-_{lbl},\ (V^-_{lbl} \cup V^+_{lbl'})$

Figure 3.8: Syntax-based system of equations defining the variable bindings before and after the execution of an expression. We let $V \backslash x = V - \{(x, b) \mid b \in \mathbf{Bindings}\}$.

For each *labeled* expression $[e]_{lbl}$, we define two sets of variable bindings V^-_{lbl} and V^+_{lbl} as the least fixed point solution to the set of equations defined with regard to the syntax of each method shown in Figure 3.8. We will primarily use binding context analysis to get the values that may be bound to a local variable. We thus let $B\ [x]_{lbl}$ be shorthand for $B\ [x]_{lbl}\ V^-_{lbl}$.

Chapter 4

An Object-Oriented Effects System

To identify the program state that is intended to be shared, the programmer must be able to name sections of program state in general. 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. Our approach to identifying state extends our object-oriented effects system, originally described in [CBS98, GB99, GS02]. Effects systems, originally studied by Gifford and Lucassen [GL86], answer the question *what state is affected*. This question must be answered, for example, to ensure all accesses to shared state are identified so that locking policies will be complied with. An effects system is an adjunct to a type system and includes the ability to infer the effects of a computation, to declare the permitted effects of a computation, and to check that the inferred effects are within the set of permitted effects. The effects of a computation include, for example, the reading and writing of mutable state.

Our object-oriented effects system uses program annotations to identify state hierarchically, to declare the upper bound of the effects of methods, and to aggregate the state of collaborations of objects. These annotations capture design intent with respect to how the programmer thinks about the state of an object, or set of objects, and the operations of those objects. Composable analyses determine effects and check that the effects of method implementations are consistent with their declared effects. Declaring the permitted effects of a method necessarily constrains the implementation of a method and any method that overrides it. Such effects declarations pose two abstraction problems: (1) we do not want effect declarations to reveal specific implementation details and (2) we want overriding implementations to be able to access, within reason, newly declared state that is necessarily unavailable to the original declaration. Thus, one of the requirements for a useful effects system for an object-oriented language such as Java is that it preserve the ability to hide the names of private fields: it should use abstract names that map to multiple mutable locations. We use named *regions* in an object: the regions of an object provide a hierarchical covering of the notional state of the object. The definition of the region hierarchy thus answers the question *what is the program state*. Programmer-declared effects in annotations and analysis-reported effects are in terms of regions, meeting our abstraction requirement.

In brief, our 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*—extrinsic identifiers of regions—and enables comparison of effects to determine if they *conflict*: if at least one effect is a write and they affect potentially overlapping targets. Our analyses are composable, *e.g.*, a method body can be checked against its annotation using the annotations on the methods it calls. Similar to type-checking in the context of separate compilation, if all method bodies are checked at some point, the program as a whole obeys its annotations. To preserve abstraction, an effects declaration cannot refer to regions that are less visible than the method being annotated. To enable modular reasoning, a reimplementation 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. To describe the state of collaborations of objects, the region framework is extended to allow state aggregations. Such aggregations allow portions of state of one object to be treated as state of another object. As a result, a complex of numerous objects, such as a linked list, can be handled as a single notional object comprising a few regions of mutable state.

We use regions and effects (1) to associate locks with state, see Chapter 5, (2) to make verification of uniqueness annotations, described in Section 4.6, composable, and (3) to support program transformation, see Chapter 8. In particular, our effects system was originally developed to support semantics-preserving program manipulations on Java source code: many transformations change the order in which computations are executed. Assuming no other computations intervene and that each computation is single-entry single-exit, it is sufficient to require that the effects of the two computations do not *interfere*: one computation does not write state that is read or written by the other.

In the following we first describe in more detail regions, targets, and effects. We then describe two extensions to our model of state to incorporate aggregations of objects: (1) incorporating the state of a uniquely referenced object into the state of the object the that refers to it, and (2) parameterizing classes by regions to allow part of the state of an object to be incorporated into the state of another object. A discussion of soundness issues follows. After a review of related work, we present a formalization of the effects system in the context of FLUIDJAVA.

4.1 Regions Identify State

To help answer *what is the program state*, the programmer defines names for extensible groups of mutable fields called *regions*. Publicly visible regions represent the abstract data manipulated by the abstract operations of the class. The read and write effects of a method are reported with respect to the regions *visible to the caller*. In this section, we describe the general properties of a region, and how regions are specified by the programmer.

4.1.1 The Region Hierarchy

The regions of a program are a hierarchy: at the root of the hierarchy there is a single region `Object.All` that includes the complete mutable state of the program. In Java, this region includes all the mutable fields of heap-allocated objects, instance variables, plus all the mutable `static` fields, class variables, of loaded classes. At the leaves of the hierarchy are all the mutable fields which again comprise the entire mutable state of the program.[1] Thus we see that each field is itself a childless region. We call the non-field regions *abstract regions*.[2] A region name is identified with the class in which it is declared, just as regular Java fields and methods are identified. A region name should not be confused with a specific region of an object; see the discussion of *targets* in Section 4.2.

The programmer declares new regions using class-level program annotations of the form

`@region` *visibility* [`static`] *region* [`extends` *parentRegion*]

This annotation declares a new region *region* in the annotated class that extends the region *parent-Region* that must be visible within the annotated class. The new region may be optionally `static`, associating a single new region with the class itself. A non-`static` declaration defines a set of regions, one for each instance of the class. The standard Java visibility modifiers apply to region declarations and retain their usual meaning. If the parent region is not explicitly named then it is

[1]External state such as file state is beyond the scope of this work. Suffice it that special regions under `All` can be used to model this external state.

[2]We use the term *abstract* to emphasize the implementation hiding characteristics of these regions, and not in the usual sense of Java `abstract` methods which impose implementation requirements on subclasses.

the `static` region `Object.All` for `static` regions and `Object.Instance` for non-`static` instance regions. These defaulting rules apply to unannotated field declarations—region declarations in their own right—as well.

A field may be placed into a programmer-defined region using the annotation

`@mapInto` *parentRegion*

To preserve the tree nature of the region hierarchy, the parent of a `static` region must be another `static` region, otherwise the region would in effect have multiple parents. An instance region may have a `static` region as its parent.

As alluded to already, there are several predefined regions. Specifically, they are

- The `static` region `Object.All`—the root of the region hierarchy. This region is the default parent of programmer-declared `static` regions, and the ultimate ancestor of every region. Unannotated methods are treated as though they have the effect `@writes Object.All`—that is, they are treated as though they might modify anything.
- The region `Object.Instance`, a subregion of `All`. Region `Instance` is the default parent of programmer-declared instance regions. Instance regions are not required to have `Instance` as an ancestor.
- The region `Array.[]`, a subregion of `Instance`. In Java, arrays are objects, so we model them as instances of a pseudo-class `Array` with the region `[]`, pronounced *element*.[3] When a subscripted element of an array is accessed, it is treated as an access of region `[]`.

4.1.2 An Example

The class `Point` in Figure 4.1a demonstrates the use of region annotations. The annotations declare a new `public` abstract region `Position` into which the `private` fields `x` and `y` are located. An instance *p* of class `Point` has four instance regions: *p*`.Instance`, which contains *p*`.Position`, which in turn contains *p*`.x` and *p*`.y`. The region `Instance` of *p*—and of all objects—is contained in the `static` region `Object.All`.

The class `Point` has methods `getX`, `getY`, and `scale` (among others). The getter methods are correctly annotated with `@reads Position` and `@writes nothing`, and `scale` is correctly annotated with `@reads nothing` and `@writes Position`.[4] Effects annotations are more fully

[3]More precision could be obtained by distinguishing the different types of arrays, especially arrays of primitive types.
[4]Our annotations are consistent with standard Java naming rules and provide for omitting `this` and the class name

```
/**
 * A two-dimensional point.
 * @region public Position
 */
public class Point { ...
  /** @mapInto Position */
  private int x;
  /** @mapInto Position */
  private int y;

  /**
   * @reads Position */
   * @writes nothing
   */
  public int getX() { return x; }

  /**
   * @reads Position */
   * @writes nothing
   */
  public int getY() { return y; }

  /**
   * @reads nothing
   * @writes Position
   */
  public scale(int sc) {
    x *= sc;
    y *= sc;
  }
}
```

(a)

```
/**
 * A colored point.
 * @region public Appearance
 */
public class ColorPoint extends Point { ...
  /** @mapInto Appearance */
  private int color;
}

/** A three-dimensional point. */
public class Point3D extends Point { ...
  /** @mapInto Position */
  private int z;

  /**
   * @reads nothing
   * @writes Position
   */
  public void scale(int sc) {
    super.scale(sc);
    z *= sc;
  }
}
```

(b)

Figure 4.1: Annotated classes (a) `Point`; and (b) `ColorPoint`, and `Point3D`. Annotations are shown in boldface.

described in Section 4.3, but we introduce them here to demonstrate the flexibility in state identification enabled by the region hierarchy. In general, method declarations are annotated with `@read` and `@write` annotations that list the portions of state that may be affected by executing the method. Returning to our example, effects in terms of `x` and `y` should not be used because the regions `x` and `y` are `private` but the methods are `public`. Using an annotation such as `@writes x, y` breaks abstraction by revealing private implementation details to clients of the class and thus is disallowed. The annotations `@writes Instance` and `@writes All`—the default annotation—are also possible, and while they do not break abstraction, they are less precise, as described in the continuation of this example below.

from region identifiers.

4.1.3 Regions and Subclasses

Regions are inherited by subclasses, subject to their visibility, and a subclass may declare its own regions, possibly within an inherited abstract region. Being able to extend existing regions is critical to being able to specify the effects of a method and later being able to meaningfully override the method. 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. The programmer thus produces a region hierarchy, which, as alluded to already, is key to the scalability of our approach by permitting the programmer to describe the state accessed by a computation with varying degrees of precision.

Consider the class `ColorPoint` in Figure 4.1b that inherits from `Point`, and declares field `color` to be in a new abstract region `Appearance`, implicitly a subregion of `Instance`. If `ColorPoint` overrode `scale`, the overriding method would not be allowed to access `color`, that is, an analysis to assure consistency between annotated effects and implementation would mark the method as erroneous. This is because `color` is not in `Position`, which is reasonable because color is not a position-related property. If `scale` were annotated with the less precise effect `@writes Instance`, the limitation would be lifted, because `color` is indeed in `Instance`. At the extreme, if the method were annotated with `@writes Object.All` then the implementation would be allowed to modify anything. The benefit of a more precise annotation is greater documentation of programmer design intent, *i.e.*, of what state the method is intended to affect, and thus enables greater assurance that an implementation is compliant with programmer intent. For example, the client of a compliant implementation of `ColorPoint` or a subclass thereof can rely on the fact that invoking `scale` is not going to change the color of a point.

We can create a second subclass of `Point`, `Point3D`, also in Figure 4.1b. In this case, the class adds an additional field `z` to the inherited `Position` region. The new implementation of `scale` is actually allowed to access more state than the original implementation because the `Position` region of `Point3D` objects is expanded to include the additional field. The implementation of `scale` thus remains consistent with the original effects specification, while at the same time it is able to expand its affected state appropriately.

4.2 Targets and State Aliasing

In Java, the handle on an object is a reference. Therefore, it is not possible to name a region of a particular object, only to name a region of the object referenced by a particular expression. *Targets*, already alluded to, are an extrinsic syntactic mechanism to name *references* to regions. If variable `p2d` refers to a `Point` object, then target `p2d.Position` 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. Targets are used in annotations when it is necessary to name state. For example, in Figure 4.1, the declared effect of method `scale` is that it writes the target `this.Position`. This target identifies the `Position` region of the object referenced by the method's receiver. That the receiver may refer to objects of different classes is what allows the target to identify more or less state for effects checking purposes. The effect of an expression is also reported in terms of the targets that may be affected.

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 `point1.Position` is distinct from the region identified by target `point2.Position` only when `point1` and `point2` are un-aliased. Similarly, aliasing can cause a single target to resolve to multiple possible regions. This ambiguity motivates our use of uniqueness annotations—see Section 4.6—to enable alias analyses to be less conservative [Boy01a]. The results of target comparison in the absence of subclasses are not violated by the introduction of subclasses. This is because of the region hierarchy and our requirement that subclass definitions respect the effects declarations in superclasses.

4.2.1 Kinds of Targets

More generally, we distinguish between four forms of targets: (1) local targets, (2) instance targets, (3) static targets, and (4) any-instance targets.

- A *local target* identifies the state associated with a local variable or method parameter. This state contains the value of the variable, which is either a primitive value, *e.g.*, an integer, a boolean, *etc.*, or a *reference* to an object, but, in Java, never an object itself. Local targets are used when reporting the effects of expressions within a method. A local target is simply the name of the local variable.

- An *instance target* identifies a region of an object. As described above, it is impossible in Java to name a specific object; it is only possible to give an expression that evaluates to the reference to that object. Because many different expressions may evaluate to the same reference, instance targets may be aliased. An instance target resembles a standard Java field reference expression, except that abstract region names are allowed to be referenced as well as field names. For example, if the expression `foo.getPoint()` returns a `Point` object, then the effect of invoking `foo.getPoint().scale(3)` is to write the region identified by the target `foo.getPoint().Position`. The expression in the target is taken *in context*, as described in Section 3.3, and thus the target identifies regions based on the objects that the expression may evaluate to at a particular point in the program.

- A *static target* identifies a `static` region. Unlike an instance region declaration, which identifies regions across all instances of a class, a `static` region declaration identifies exactly one region associated with the class itself. Thus the region identified by a `static` target can never be ambiguous and a static target is simply the fully qualified name of a `static` region, *i.e.*, the region name plus the name of the class it is declared in, as in `Object.All`.[5]

- An *any-instance target* identifies state based on the name of an instance region, refined by a class that constrains the set of objects to be considered. An any-instance target of the form `any(`*class*`)`.*region* identifies all the regions *region* of all the possibly shared objects of class *class*. In particular, the target does not identify the *region* region of any object that is known to be uniquely referenced. For example, the target `any(Point).Position` identifies all the `Position` regions of all `Point` instances, including `ColorPoint` and `Point3D` instances. Whereas the target `any(Point3D).Position` identifies all the `Position` regions of all the `Point3D`—but not `Point`—instances.

4.2.2 Targets and Method Effects

The purpose of the any-instance target requires additional explanation. They solve a problem that occurs when specifying the effects of a method. The declared effects of a method must be in terms of state identifiers—targets—that are guaranteed to refer to the same state throughout the execution of the method. Otherwise the declared effects will fail to account for all the effects of the method, and the effects system will be unsound. Local targets make no sense in the context of method effects: the method cannot affect the local state of the caller, and the local state of the executing method disappears once the method returns. Static targets refer to unambiguous state, and are therefore acceptable in method effect declarations. Instance targets are problematic because they are in terms of a general object-valued expression. Consider the target `o.f.region`: if the value in field `f` changes during the method's execution, the caller can only account for the regions that the target may have referred to when the method was invoked—it has no information about what may have been assigned to the field—and therefore the declared effects will not account for all the

[5] Or more pedantically, `java.lang.Object.All`.

possible effects of the method execution. Spoto and Poll demonstrate the problems that occur in this situation in the context of the `assignable` clause of the Java Modeling Language (JML) [SP03]. In particular, there is little to be gained from the ability to declare more precise effects because they are difficult to reason about and must generally be expanded into a set of less precise effects.

Because of these difficulties, we restrict instance targets in effect declarations to be of the form *p*`.`*region*, where *p* is a method parameter or the receiver `this`. That said, the declared effects of a method must still account for the state affected by expressions such as

```
this.f1.f2 = ...
```

that appear within a method implementation. When reasoning about effects within the method implementation, the target `this.f1.f2` is useful, but as just described, it cannot be used in the declared effects. Our effects system thus contains any-instance targets which identify regions based on classes. They are easier to reason about than the alternatives, and, in practice, no less precise.

4.3 Effects

We distinguish between two kinds of effects: *read effects*, those that may read the contents of a region; and *write effects*, those that may change *or read* the contents of a region. As seen in Section 4.1.2, the programmer may annotate a method with its permitted effects; it is an error for any implementation of the method to have more effects than are declared. The soundness of static analysis of the effects system would otherwise be compromised. We emphasize that effects are permissive: an effect describes the upper bound on how state might be affected. Similarly, if a region is affected by an expression, is it possible that only part of the region is affected. For example, invoking the method `Point.getX` in Figure 4.1 has the effect `@reads Position` even though the implementation actually affects only the field `x`. Because effects are permissive, a write effect does not guarantee that a variable is written. Thus there would be no data-dependency purpose served by not permitting a write annotation to give permission to read—it would only make the annotations more verbose.

Effect annotations are introduced in Figure 4.1. In general, effects are specified using the annotations

`@reads` $target_1$, ..., $target_n$

and

`@writes` $target_1$, ..., $target_n$

where *target* is restricted to be a static target, an instance target identifying a region of a method parameter (including `this`), an any-instance target, or the special target `nothing` that identifies no regions. Section 4.11.2 describes this restriction more formally. As mentioned previously, our implementation allows `this` to be omitted in targets, thus `@reads Instance` is the same as `@reads this.Instance`. Unannotated methods are assumed to have the effect `@writes Object.All`. The burden of annotating effects is lightened by allowing a partial specification of effects: if a `@reads` annotation but not a `@writes` annotation is present, then the annotation `@writes nothing` is assumed, and similarly for a missing `@reads` annotation.

4.3.1 Computing Effects

Effects are computed in a bottom-up traversal of the syntax tree. Every statement and expression has an associated set of effects. In general, an effect is produced by any expression that reads or writes a mutable variable or mutable object field; in Java, immutability is indicated by the `final` modifier in field and variable declarations. More specifically, a write effect directly originates from an assignment expression, which in Java includes pre- and postfix increment and decrement operators, or indirectly from a call to a method/constructor with a write effect. A read effect directly originates from a variable use or a field reference expression, or indirectly from a call to a method/constructor with a read effect. Recall that array subscripting is treated as a reference to the region `[]`.

The effects of expressions and statements include the union of the effects of all their subexpressions. The targets in the computed effects use the most specific regions possible. Thus, for example, analysis of the expression `this.x *= sc` from the `Point.scale` method determines that local target `this` is read, instance target `this.x` is written, and local target `sc` is read. A more precise description of how effects are computed for FLUIDJAVA is given in Section 4.11.2.

The analysis is intraprocedural; effects of method calls are obtained from the annotation on the called method, rather than from an analysis of the method's implementation. Because the method's declared effects are with respect to the formal parameters of the method, the effects of a particular method invocation are obtained by substituting the actual parameters for the formal parameters in the declared effects.

4.3.2 Comparing Effects

Assuming the effects system is sound, see Section 4.9, two expressions *interfere*, that is, have a data dependency between them, only if they have conflicting sets of effects. Two effects *conflict* if at least one is a write effect and they affect targets that describe state that may overlap, that is, may refer to the same mutable state at runtime. Two targets may overlap only if they refer to overlapping regions, and the hierarchical nature of regions insures that regions overlap only if one is included in the other. Furthermore, an instance target can only overlap another instance target if the objects they refer to could be identical. The details of our implementation are described in Section 4.11.4.

4.3.3 Checking Declared Effects

The effects of a method implementation must be checked against the declared effects of the method. Every effect that might be to state that exists prior to the method or constructor being called must be accounted for by the declared effects. Some implementation effects can be ignored or "masked," because they are known to affect state that cannot be observed by the caller:

- Effects on local targets are masked because they are irrelevant outside of the method's stack frame.
- Effects on newly created objects are masked because they are imperceivable outside of the method body. In particular, the state did not yet exist in the calling context.
- For a constructor, effects on the object being constructed can be ignored. Again, this is because the object did not exist prior to the constructor being called.

Checking that implementation effects are consistent with the declared intended effects is complicated by the limitations discussed in Section 4.2.2: it only makes sense for the declared effects to use any-instance targets or instance targets restricted to refer to regions of the objects referenced by the method's parameters (including the receiver this). Effects inferred from the method's implementation, however, are in terms of more general targets. We need to be able to determine if an implementation effect is "covered" by the declared effects. Because any-instance targets cause effect conflict results to be overly conservative, we would like to maximize the number of effects that can be determined to be covered by an effect on a region of an object referenced by a method parameter. We define a general relation covers that relates a set of formal parameter names to an expression from a method. For a method or constructor m, let e be an expression from the method, F_m be the set of formal parameters of the method, $orig_f$ refer to the object bound to the formal $f \in F_m$ on entry to the method m, and N be the set of objects newly allocated during execution of the method. For a set $F \subseteq F_m$, we say F covers e if, for any call of the method, the objects that e may refer to during the course of executing the method, O, are such that $O \subseteq N \cup \{orig_f \mid f \in F\}$. That is, F covers e if each object that e might refer to that is not a newly allocated object can be proved to be equal to $orig_f$ for some $f \in F$. A simple approximation of the covers relation is

$$\begin{cases} \{f\} \text{ covers } f & \text{where } f \text{ is the use of a final formal} \\ \{\} \text{ covers } (\text{new } c) & \end{cases}$$

Clearly, a write effect must be accounted for by a write effect. Because write effects also give "permission" to read, however, a read effect can be accounted for by either a read or a write effect. We thus now discuss informally how the targets describing the affected state are accounted for. In general, the region hierarchy allows targets at a coarser granularity to account for targets identifying state at a finer granularity. Accounting for static targets and any-instance targets is thus straightforward. As mentioned above, local targets are masked. Instance targets are also straightforwardly accounted for by static and any-instance targets. The challenge is to account for an effect on an instance target *e* **.** *region* with a declared effect on an instance target. For this we rely on the covers relation. If *e* is a method parameter, then it is may be accounted for by a declared effect on a method parameter, according to the region hierarchy. Otherwise, if there exists a set F such that F covers e, then the target may be accounted for by instead accounted for *all* the instance targets *f* **.** *region*, where *f* is an element of F.

4.4 Example: Class `AtomicInteger`

We now use several examples to demonstrate further the use of region and effect annotations, and to highlight how the design of the region hierarchy is a reflection of programmer design intent. Our first example is an "atomic integer" class: an encapsulated integer value that is synchronized for concurrent use. Such classes exist in popular concurrency libraries, *e.g.*, `SynchronizedInt` in `util.concurrent` [Lea]. The source code for class `AtomicInteger` is shown in Figure 4.2. Objects of the class are wrappers around an `int` field; the methods provide `get`, `set`, and `testAndSet` access to the value. Access is synchronized by using an object as a lock to protect the data. This lock may be the `AtomicInteger` object itself, or some other object that is identified when the object is constructed.

The state of an `AtomicInteger` consists of the two fields `lock` and `value`. These have been left unannotated, which by default makes them subregions of `Instance`. The field `lock` is `final` to prevent the identity of the lock from changing. Both constructors are annotated to have no effects. This is correct because the constructors only affect the object's `Instance` region, via the `value` region, so the effects are masked. The `getLock` method is also declared to have no effects. This

```
public class AtomicInteger {
  /** Object to use as the mutex to protect <code>value</code>. */
  private final Object lock;
  /** The encapsulated integer value. */
  private int value;

  /**
   * Create an integer that uses a specific lock.
   * @writes nothing
   */
  public AtomicInteger( int val, Object l ) {
    lock = l;
    value = val;
  }

  /**
   * Create an integer that uses itself as the lock.
   * @writes nothing
   */
  public AtomicInteger( int val ) { this( this, val ); }

  /**
   * Get the lock used to protect this object.
   * @writes nothing
   */
  public Object getLock() { return lock; }

  /**
   * Get the integer value.
   * @reads Instance
   */
  public int get() { synchronized( lock ) { return value; } }

  /**
   * Set the integer value.
   * @writes Instance
   */
  public void set( int val ) { synchronized( lock ) { value = val; } }

  /**
   * Set the value to <code>newVal</code> only if
   * the current value is <code>assumedVal</code>
   * @return <code>true</code> if the new value was set.
   * @writes Instance
   */
  public boolean testAndSet( int assumedVal, int newVal ) {
    synchronized( lock ) {
      boolean success = (assumedVal == value);
      if(success) value = newVal;
      return success;
    }
  }
}
```

Figure 4.2: Source code for `AtomicInteger` with effects annotations. Annotations are shown in boldface.

declaration is correct even though the implementation of the method clearly accesses a field. The method body has a single effect: it reads local target `this` when referencing the field `lock`. Reading the field `lock` produces no effect because the field is `final`. The effect on the local target is maskable so the declared effects are correct.

The last interesting method is `testAndSet` which is declared to have the effects `@reads nothing` and `@writes Instance`. After masking, the computed effects of the body the effects `reads this.value` and `writes this.value` remain. Because a write effect includes the possibility of reading, both effects are included in the method's declared effects.

4.5 Example: Class `BoundedFIFO`

A more sophisticated example is the class `BoundedFIFO`, originally introduced in Section 1.3, from the Log4j logging library shown unannotated in Figure 1.2. The class implements a circular queue using an array `buf`, with head and tail indices `first` and `next`. The buffer is non-blocking: `get`ing from an empty buffer returns `null`, and `put`ing to a full buffer silently drops the new event. The class is intended to be used for communication between threads. Even though it does not contain any synchronization code, it turns out clients of `BoundedFIFO` objects are expected to acquire the lock on the FIFO object. In Chapter 5 we show how to document this design intent. Client-side blocking is facilitated by the methods `isFull`, `wasFull`, and `wasEmpty`.

The state of a `BoundedFIFO` consists of some counters that track the size of the buffer and the head and tail pointers plus the array reference `buf`. Once again, because there are no annotations, all the fields are assumed to be subregions of `Instance`. Because the methods and constructors are unannotated they are treated as if they had the declared effect `@writes Object.All`. All the method implementations are, of course, compatible with this. But reasoning about the class using effects of this granularity is not very useful. Also, *without annotations we lose programmer design intent*. Ideally, we would like to be able to declare that each method affects at most the region identified by the target `this.Instance`, that is, it only affects the receiving object.

It is tempting, for someone who understands regions, to subdivide `Instance` into two regions, the first containing the fields `buf`, `first`, and `next`, and the second containing the fields `numElts` and `size`. This is reasonable because the "getter" methods `getMaxSize`, `length`, `wasEmpty`, `wasFull`, and `isFull` do not affect the fields in the first group, so they could be given effects declarations that indicate they only affect a portion of the object. This partitioning of state makes the design intent more explicit than leaving all the fields as children of `Instance`. In particular, that only `get` and `put` should affect the contents of the buffer. However, it does not assist in reasoning about how uses of `BoundedFIFO` objects may conflict because (1) all the getter methods have read effects only and thus never conflict with each other, and (2) `get` and `put` potentially write to both sets of fields so their use would still conflict with uses of the getter methods. Herein, we decide in favor of brevity of annotation and do not subdivide the `Instance` region.

4.5.1 Annotating BoundedFIFO

Our first attempt at annotating the effects of `BoundedFIFO` is shown in Figure 4.3, which also introduces our notation for easing the annotation of similar methods. The class-level annotation

`@methodSet` *mset* = $method_1$, ..., $method_n$ [, ...]

identifies a set of methods,[6] where *mset* is an identifier not already used as the name of a method or method set. The optional trailing ellipsis indicates that the specification of the set is incomplete and that methods may add themselves to the set using the method-level annotation

`@inSet` *mset*

An entire set may be annotated using the class-level annotation

`@set` *mset annotation*

where *annotation* is a method-level annotation (without the initial "`@`").[7] Thus the annotations in Figure 4.3 succinctly describe the intent that the methods `getMaxSize`, `length`, `wasEmpty`, `wasFull`, and `isFull` all have the effect `@reads Instance`.

As discussed above, we would like to annotate methods `get` and `put` with `@reads nothing` and `@writes Instance`. Unfortunately, if we were to do so, our effects checker would report that the declared effects do not account for all the effects of the implementation. This is because these two methods *do* affect an object that is not the FIFO object: the array object referenced by the field `buf`. Expressions such as "`... = buf[first]`" affect the region `[]` of the object referenced by the expression `this.buf`. The declared effects of the `get` and `put` methods must include these effects. According to the discussion in Section 4.2.2 we are forced to annotate that the `get` and `put` methods can affect the target `any(Array).[]`, *i.e.*, that they can affect any array. These annotations are not ideal:

[6] Overloaded names are disambiguated via explicit identification of their parameter types, *e.g.*, `println(Object)` *vs.* `println(int)`.

[7] A set may be added to another set using the annotation template `@set` S_1 `inSet` S_2. The name of a method set may also appear to the right of the "=" in a `@methodSet` annotation.

```
/**
 * @methodSet readers = getMaxSize, length, wasEmpty, wasFull, isFull
 * @set readers reads Instance
 */
public class BoundedFIFO { ...
  LoggingEvent[] buf;
  int numElts = 0, first = 0, next = 0, size;

  /** ...
   * @writes nothing */
  public BoundedFIFO(int size) { ... }

  /** ...
   * @writes Instance
   * @reads any(Array).[]
   */
  public LoggingEvent get() { ... }

  /** ...
   * @writes Instance, any(Array).[] */
  public void put(LoggingEvent o) { ... }
}
```

Figure 4.3: Our first attempt at annotating class `BoundedFIFO`. Notice the exposure of the array effects for methods `get` and `put`. Annotations are in boldface.

- From the point of view of reasoning about the effects of the class, the effects of `get` and `put` are surprisingly large. The effects are such that `put`ing an item to any `BoundedFIFO` instance will interfere with `get`ing an item from any other `BoundedFIFO` instance. This is surprising because there is no reason to believe that distinct FIFO instances share state.
- From the point of view of encapsulation, the effects still leak implementation related details to clients of the class. Namely, that `BoundedFIFO` is implemented using an array. This detail ought to be irrelevant to clients.
- From the point of view of *design intent*, the annotations fail to capture the intent that the array referenced by the field `buf` is *part of* the FIFO object. This intent is fundamental to how the programmers think about the class, and is why the previous two points are surprising to both clients and maintainers of the class.

4.6 State Aggregation through Uniqueness

As the example above shows, the state of a single design abstraction often spans the state of multiple instantiated objects. This can interfere with the clarity of expressed design intent and force the principles of encapsulation to be violated. We have two techniques for aggregating state from many objects into a single region. The first is based on the exploitation of unaliased references, also known as unique or unshared references. The second technique, presented in the next section, is based on parameterizing classes by regions.

A field in Java contains only a reference to an object. Thus a region does not contain the state of other objects, only references to other objects. The intent for class `BoundedFIFO`, however, is that the state of the FIFO also include the entirely separate array object referenced by the field `buf`. In Section 4.5 we discuss some of the problems that result from the failure to capture this design intent. The programmer is inclined to think that the state of the array is part of the FIFO object because of the additional intent that the array referenced by `buf` is not intended to be shared with other objects. That is, the reference is meant to be *unique*. State of a uniquely referenced object may be considered to be state of the referencing object. In particular, regions of the referenced object are mapped back into regions of the referencing object. This mapping can be performed only for uniquely referenced objects. Consider the case of an object referenced by two different objects that both map its state into their own. Suppose that the two referring objects are visible in the same scope via variables `p` and `q`, and alias analysis is able to determine that they are not aliased. Because effects on our shared object through `p` are now treated as effects on `p` and because effects on our shared object through `q` are now treated as effects on `q`, we can falsely conclude that two writes to the same region of our shared object do not conflict.

We use the field-level annotation

`@unshared`

to record explicitly the design intent that the field has no aliases at all when it is read, that is, any object read from it is not accessible through any other variable. A separate composable static analysis, described in [Boy01a], verifies that unshared fields are properly used. The analysis also supports `@unshared` method parameters and return values, and `@borrowed` parameters—variables for which no new aliases may be created. Further discussion on these features is beyond the scope of this work, although we rely on the design intent they express for several of the analyses we present herein. Once a field has been identified as unaliased, it is possible to treat the object referenced by that field as part of the referring object. The instance regions of a uniquely referenced object may be *aggregated* into regions of the referring object using the field-level `@aggregate` annotation:

`@aggregate` s_1 `into` d_1`, ...,` s_n `into` d_n

This declares the intent that region s_i of the uniquely reference object is aggregated into the region d_i of the referring object. That is, effects on region s_i of the referenced object are considered to be effects on region d_i of the referring object.

Two regions related by the region hierarchy must not be mapped into unrelated regions. Suppose

```
/**
 * @methodSet readers = getMaxSize, length, wasEmpty, wasFull, isFull
 * @set readers reads Instance
 */
public class BoundedFIFO { ...
  /**
   * @unshared
   * @aggregate Instance into Instance
   */
  LoggingEvent[] buf;

  /** ...
   * @writes Instance */
  public LoggingEvent get() { ... }

  /** ...
   * @writes Instance */
  public void put(LoggingEvent o) { ... }
}
```

Figure 4.4: Class `BoundedFIFO` annotated to use uniqueness aggregation. The effects of `get` and `put` no longer reveal array effects. Annotations are in boldface.

(1) region `p` is a subregion of `r` in an unshared object; (2) they are mapped into regions `a` and `b` of the referencing object, respectively; and (3) neither `a` nor `b` is a descendent of the other. A write to region `p` of the unshared object should conflict with a write to region `r`, but via mapping we can falsely conclude that they do not conflict because regions `a` and `b` of the referencing object are disjoint. We additionally require the mapping to respect the ancestor–descendent relationship to prevent potential logical inconsistencies and unsoundness in the formalization of the semantics of the effects system. Specifically, if region `r` of unshared field `u` is aggregated into region `q` of the referring object, then any descendent of `r` must be aggregated into a descendent of `q`, or `q` itself. We also require that all regions of the uniquely referenced object must be mapped into some region of the referring object. That is, for a uniquely referenced object declared to be of class `c`, we require that for each region in `c` there exists a superregion of `c` that is explicitly mapped into the referring object.

Returning to class `BoundedFIFO` we can use annotations to capture the design intent that the array referenced by `buf` is meant to be part of the FIFO object itself. The field is declared to be `@unshared` and the `Instance` region of the referenced array, which includes the `[]` region, is aggregated into the FIFO's `Instance` region; see Figure 4.4. Static analysis assures that `buf`'s use is consistent with being unshared. Effects analysis is now able to determine that `put` and `get`'s effects to the array are included in effects to the FIFO object itself. Both methods can thus be annotated with the more precise effect `@writes Instance`.

When checking that an implementation's effects are consistent with the method's declared effects, effects on `@unshared` parameters can be ignored. Because the method's caller must "give up" access to such an object, it cannot observe any future changes to that object's state. Additionally, the aggregation mappings of `@unshared` fields are consulted to reinterpret effects on a referenced object into effects on the referencing object, which allows effects that would otherwise have to be accounted for via any-instance targets to be accounted for by effects on regions of the method's parameters (including `this`).

4.7 State Aggregation through Parameterization

A second kind of aggregation is achieved by parameterizing class definitions by regions. This technique allows portions of an object to be aggregated into another object, and does not impose any aliasing constraints. We use a notation similar to C++ templates. Figure 4.5 shows a pair of classes from the Jigsaw open-source Java web server with annotations. Class `ThreadCache` maintains a doubly linked list of `CachedThread` objects. The annotations 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 line 2. The head and tail pointers, `freelist` and `freetail`, are declared on line 11 to be subregions of `Threads`.

To perform the intended aggregation, there must be an identification for each `CachedThread` object of which region is to contain its fields `next` and `prev`. The *region parameter* `Backbone`, declared in angle brackets on line 34, accomplishes this. Fields `next` and `prev` are made children of the region bound to `Backbone` on line 41. The parent region (bound to `Backbone`) is specified when a `CachedThread` object is instantiated. This binding conceptually occurs on line 16 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 12, 15, 20, and 42. The declaration on line 42 ensures that a `CachedThread` can refer only to other `CachedThreads` parameterized by the same region.

The parameterization of `CachedThread` enables enforceable separation between representa-

```
/**
 * @region public Threads
 * @region public CacheInfo
 */
public class ThreadCache {
  /** @mapInto CacheInfo */
  protected int threadcount, usedthreads;

  ...

  /** @mapInto Threads */
  protected CachedThread /*@<this.Threads>*/  freelist, freetail;

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

  /** @writes t.ThreadInfo, this.Instance */
  synchronized boolean isFree(CachedThread /*@<this.Threads>*/ t, ...) {
    if(!t.isTerminated()) { ... }
    else { ...
      t.prev = freetail;
      if(freetail != null) freetail.next = t;
      freetail = t;
      if(freelist == null) freelist = t;
      usedthreads--; ...
    } ...
  }
  ...
}

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

  /** @writes ThreadInfo */
  synchronized boolean isTerminated() { ... }

  ...
}
```

Figure 4.5: Annotated versions of `ThreadCache` and `CachedThread`.

tions of distinct `ThreadCache` aggregations, 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.

In Section 5.1, 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 $region_1$, ..., region $region_n$> */

to a class definition. Uses of the class are parameterized by appending to the use the annotation

/* @< $target_1$, ..., $target_n$> */

4.8 Java Interfaces

Our effects system must accommodate additional complexities of the Java language if it is to be adoptable on real programs. In particular, to be more broadly applicable to production code, our effects system must incorporate Java interfaces, which only declare methods but never provide implementations. A class may implement multiple interfaces. To permit useful annotations on method headers in interfaces, we must be able to add abstract regions to interfaces and thus we must handle multiple inheritance of instance regions. Static regions, like static fields, are not inherited and thus do not complicate matters. Multiple inheritance of instance regions is handled by permitting a class or interface inheriting regions from an interface to map them into other regions as long as the region hierarchy is preserved. As with the mapping of regions in uniqueness aggregations, any two regions, both of which are visible in a superclass or superinterface of the classes or interfaces performing the mapping, must have the same relationship when mapped to regions in the inheriting class as they do in their original class or interface. Conflicting relations in superclasses or superinterfaces may forbid certain inheritance combinations. We have not yet formalized or implemented this aspect of the effects system.

4.9 Towards Soundness

An important aspect of a static effect system is for it to be *sound*, that is, it must not say two computations do not interfere through the access of some shared state when in fact they do. Proving soundness requires a runtime definition of interference which we must then show is conservatively approximated by our analysis. The runtime state of the program we are interested in includes fields of reachable objects as well as the state of local variables and temporaries in all activation frames. The effects of a computation can then be seen as reads and writes on elements of the state. These effects can then be compared to that of the second computation. Interference occurs when an element of state is written by one computation and read or written by the other.

We have not yet formulated such a soundness proof. The primary impediment is the development of a type-system and state semantics that adequately incorporates unshared references and the state aggregation they enable. Even though we present here our effects system as a client of our system for assuring unique references, the two systems are in fact interdependent. As we have discussed herein, the effects system uses uniqueness to enable state aggregations. However, assuring uniqueness requires effects, in particular to assure that unique references are not read when borrowed references are live. This interdependence is described in more detail in [Boy01b]. The interdependence of the two systems means that they cannot be proven sound independently. A satisfactory combination of uniqueness and effects has not been developed. Capabilities, also known as permissions, are presently being explored as a solution to this problem. In the realm of uniqueness, Boyland, *et al*. [BNR01] use capabilities to give a semantics for uniqueness without effects.

Permissions are useful for describing effects as well, and in fact provide a solution to a tricky analysis problem. Aliasing frustrates reasoning about effects because effects do not include enough information to perform interference checking. In our effects system, this is manifest (1) in our use of targets and (2) the need to use an alias analysis—unrelated to the unique analysis—to compare targets. Furthermore, we have a non-traditional aliasing problem that we call *MayEqual* [BG99], although it can be approximated using more traditional alias analyses: we must determine whether two expressions, each at a different point in the program, may point to overlapping sets of locations. Clarke and Drossopoulou have the same problem, but directly incorporate a simple type-based alias analysis into their effects system [CD02]. Interpreting effects as permissions to read or write memory locations avoids this problem, but early permission systems could not adequately distinguish read from writes. Boyland's "fractional permissions" [Boy03a] overcome this difficulty, and he shows how to check read and write effects with permissions, although the system does not feature uniqueness or state aggregation.

The current approach to integrating uniqueness and effects is based on "adoption and focus" as presented in the context of the Vault programming language [FD02]. This model is attractive because it combines linearity, *e.g.*, uniqueness, with permissions, but as described is not sufficient for modeling our effects system. In [Boy03b], Boyland extends the adoption and focus model with fractional permissions, multi-focus, and adoption of fields to enable distinguishing of reads from writes, aggregation of multiple objects at once, and modeling of hierarchical regions, respectively. In summary, proving the soundness of the effects system is difficult because of the interdependence of uniqueness and effects. Ongoing research, however, on the integration of permission systems, linear pointers, and regions appears to be close to producing a system upon which our particular uniqueness and effects systems can be proved sound.

4.10 Related Work

Reynolds [Rey78] showed how interference in Algol-like programs could be restricted using rules that prevent aliasing. His technique, while simple and general, requires access to the bodies of procedures being called in order to check whether they operate on overlapping global variables. The work includes a type system with mutable records, but the records cannot have recursive type; in particular, lists and trees are not possible.

Effects were first studied in the FX system [GJLS87], a higher-order functional language with reference cells. The burden of manual specification of effects is lifted through the use of effects inference as studied by Gifford, Jouvelot, and Talpin [JG91, TJ92]. The work was motivated by a desire to use stack allocation instead of heap allocation in mostly pure functional programs, and also to assist parallel code generation. The approach was demonstrated successfully for the former purpose in later work [AFL95, TT94]. These researchers also make use of a concept of disjoint "regions" of mutable state, but these regions are global, as opposed to within objects. In the original FX system, effects can be verified and exploited in separately developed program components.

Jackson's Aspect system [Jac95] uses an abstraction mechanism similar to our regions to specify the effects of routines on abstract data types. He uses specifications for each procedure to allow checking to proceed in a modular fashion. In his work, however, the specifications are *necessary* effects and are used to check for missing functionality.

Extended Static Checking for Java (ESC/Java) [FLL^{+}02, DLNS98] 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 [LBR99], are compiled into verification conditions to be proved. 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. ESC/Java does not support unique references. It is not possible to abstract state—fields—into named aggregates, nor is it possible to express state abstractions across aggregates of objects.

Leino, *et al*. [Lei98, LPHZ02] describe "data groups" as a means to abstractly identify object state. Like our regions, data groups are hierarchical sets of fields. Unlike our regions, a field or data group may be included in multiple parent groups. This is sound because they are not intrinsically interested in object-oriented effects, but in using modifies clauses on methods to check program assertions modularly. Thus their presentation of data groups does not include read effects. State aggregation akin to our uniqueness aggregation is possible via so-called "rep inclusions" in which the fields of a referenced object are mapped into the data groups of the referring object. The field that refers to the mapped object is known as a pivot field. Soundness is maintained by restricting the use of pivot fields. Like our uniqueness aggregation, these restrictions amount to controlling aliasing of pivot fields, although in our work we rely on a separately formalized and more general uniqueness analysis. One restriction "ensures that values in pivot fields are either null or unique, except possibly for copies stored in [read-only] formal parameters on the call stack." The second restriction controls how pivot fields may be passed as parameters: it prohibits the value of a pivot field from being passed to a method whose modifies clause declares that it may modify the state of the object referenced by the pivot field.

Object ownership models [CPN98] transitively aggregate the state of an object into the state of its owner. Object ownership allows aggregation only at the granularity of objects. Our region-based approach instead models what might be called *field ownership* and is not primarily concerned with the objects referenced by fields. Clarke and Drossopoulou [CD02] describe an object-oriented effects system where state aggregation is based on object ownership instead of unique references, as is used in our approach. Objects are not subdivided into regions of state, although they briefly suggest how this might be integrated into their approach. Instead effects are reported using "effect shapes" that describe sets of objects based on the object-ownership tree. The effect shape `this.1`, for example, denotes the set of objects owned by `this`, whereas the shape `under(this.2)` is the set of objects at least two steps from `this` in the ownership tree. This representation allows "the abstract representation of effects on unknown parts of an object's internals, while retaining some precision when dealing with these internals." Our use of uniqueness aggregation does not distinguish between the levels of indirection. Their effect system is thus "orthogonal" to ours: their precision is horizontal over a tree, ours is vertical. We can distinguish multiple regions of an object with transitive mapping for ownership. Their system sees only one region for all fields of an object but can distinguish how far away it is from the root object. Their effect system does not report effects on local variables—they are always immutable. Aliasing problems are not directly addressed, instead type-based disjointness is relied upon.

Boyapati and Rinard describe a ownership-based type-system to enforce locking conventions in Java [BR01]. Classes are parameterized by owners, which are provided when an object is created. A special `unique` owner identifies objects that must always be unaliased; our system is instead concerned with unaliased *variables*, which may only hold unique references. The `readonly` owner identifies immutable objects. 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.

Yates describes preliminary work on a type-and-effect system for encapsulating memory in Java [Yat99]. His system is closer to that of [LG88] than it is to ours. The state of the program is divided into disjoint regions that contain objects in their entirety. Objects may refer to objects in other regions. Read and write effects are not distinguished—an effect is just the name of a region. Yates gives a type system and semantics for an elaborated language in which methods are elaborated with the region in which their parameters and return value reside. These elaborations are not polymorphic and the process by which a program is elaborated is left unspecified. Thus the number of regions in a program, or even their origin is left unspecified. His goal is not to describe design intent or the organization of program state but to prove the correct use of encapsulated expressions, identified by a new keyword. An encapsulated expression is one that "should produce the same result from the same arguments in any context."

Bierman and Parkinson [BP03] formalize a subset of our effects system as presented in [GB99] and provide a effects inference algorithm. The formalization is proved to be sound and the algorithm correct with respect to an operation semantics of effects. The effects system they formalize does not distinguish between objects—all abstraction regions are static. Their system also lacks a region hierarchy, uniqueness, and therefore, uniqueness aggregation.

4.11 Effects in FLUIDJAVA

We now present a subset of our effects system more formally by extending FLUIDJAVAwith regions, effects declarations, uniqueness, and uniqueness aggregation. We mean to include normally declared fields when we use the term *region*. We use *abstract region* when we do not intend to include fields. Because FLUIDJAVA does not have static fields our presentation of regions does not include static regions or static targets. In particular we do not have the region `Object.All`—the target `any(Object).Instance` is sufficient to refer to all the heap state in the program. We have not

$$
\begin{array}{rcl}
\mathit{defn} & ::= & \mathsf{class}\ cn\ \mathsf{extends}\ c\ \mathit{effect}\ \{\mathit{region}^*\ \mathit{field}^*\ \mathit{meth}^*\} \\
\mathit{region} & ::= & \mathsf{region}\ \mathit{argn}\ \mathsf{in}\ \mathit{argn} \\
\mathit{field} & ::= & \mathit{shared\_field} \mid \mathit{unshared\_field} \\
\mathit{shared\_field} & ::= & \mathit{mod\_t}\ \mathit{fd}\ \mathsf{in}\ \mathit{argn} = e \\
\mathit{unshared\_field} & ::= & \mathsf{unshared}\ \mathit{mod\_t}\ \mathit{fd}\ \mathsf{in}\ \mathit{argn}\ \mathsf{aggregate}\ \mathit{agg}^+ = e \\
\mathit{agg} & ::= & \mathit{rgn}\ \mathsf{into}\ \mathit{argn} \\
\mathit{meth} & ::= & t\ mn_{lbl}(\mathit{arg}^*)\ \mathit{effect}\ \{\mathit{body}\} \\
\mathit{effect} & ::= & \mathsf{reads}\ \mathit{tgt}^+\ \mathsf{writes}\ \mathit{tgt}^+ \\
\mathit{tgt} & ::= & \mathsf{nothing} \mid x \mid e.\mathit{rgn} \mid \mathsf{any}(c).\mathit{rgn}
\end{array}
$$

The disjoint name spaces are extended with the set of abstract region names; the set of region names is the set of abstract region names together with the set of field names.

$\mathit{argn} \in$ abstract region names $\quad \mathit{rgn} \in$ field names $\cup$ abstract region names

Figure 4.6: Syntax modifications to FLUIDJAVAwith labeled expressions for regions and effects.

yet formalized region parameters, although in general they could be handled by associating a function with each parameterized class that represents the portion of the region hierarchy defined by the region parameters of that class, and by extending the environment to map formal region parameters to *targets*. The subregion relation would then be replaced with a subtargeting relation defined in terms of inferences based on the environment and relevant portions of the region hierarchy.

4.11.1 The Extended Language

The syntax of FLUIDJAVA with labeled expressions is modified as shown in Figure 4.6. Region declarations are added to class definitions, and field declarations must now include a parent region. A field may be declared to be unshared, and such fields also have a mapping from regions of the referenced object into regions of the referring object. A class declaration now contains a declaration of effects: these are the effects of constructing a new instance. In real Java, the constructors are annotated with effects just as methods are. Method declarations are extended to include a declaration of their latent effects. The general syntax for effects and targets is also shown in Figure 4.6 and defines the set of effects and targets used our in formalizations. Targets are the empty target nothing, the local target x, the instance target $e.rgn$, and the any-instance target $\mathsf{any}(c).rgn$. In Section 4.11.2, we present well formedness rules that restrict the shape of targets in effect declarations.

For convenience, we abuse notation in the following ways

- We treat members of tgt^* as sets of targets.
- We treat the empty target nothing as the empty set.

$$\begin{aligned}
&\textbf{RegionsOnce}(P) \equiv \\
&\qquad \textbf{FieldsOnce}(P) \\
&\qquad \wedge\ (\forall \mathit{argn}, \mathit{argn}' : \textsf{class} \cdots \{ \cdots \textsf{region}\ \mathit{argn} \cdots \textsf{region}\ \mathit{argn}' \cdots \} \text{ is in } P \Rightarrow \mathit{argn} \neq \mathit{argn}')
\end{aligned}$$

$$c.\mathit{rgn} \Subset^{\mathrm{c}} c \Leftrightarrow (\textsf{class}\ c \cdots \{ \cdots \textsf{region}\ \mathit{rgn} \cdots \} \text{ is in } P)\ \vee\ (\exists \mathit{mod\_t} : \langle c.\mathit{rgn}, \mathit{mod\_t} \rangle \Subset^{\mathrm{c}} c)$$

$$c'.\mathit{rgn} \in^{\mathrm{c}} c \Leftrightarrow c'.\mathit{rgn} \Subset^{\mathrm{c}} c'\ \wedge\ c \leq^{\mathrm{c}} c'$$

$$\mathit{rgn} \overset{c}{\hookrightarrow} c'.\mathit{rgn} \Leftrightarrow \exists c' : c'.\mathit{rgn} \in^{\mathrm{c}} c$$

$$c.\mathit{rgn} \prec^{\mathrm{r}} c'.\mathit{rgn}' \Leftrightarrow \textsf{class}\ c \cdots \{ \cdots \mathit{rgn}\ \textsf{in}\ \mathit{rgn}' \cdots \} \text{ is in } P \wedge c'.\mathit{rgn}' \in^{\mathrm{c}} c$$

$$\leq^{\mathrm{r}}\ \equiv \text{ the transitive, reflexive closure of } \prec^{\mathrm{r}}$$

$$\mathit{rgn} \leq^{\mathrm{r}}_{c,c'} \mathit{rgn}' \Leftrightarrow \mathit{rgn} \overset{c}{\hookrightarrow} \hat{c}.\mathit{rgn} \wedge \mathit{rgn}' \overset{c'}{\hookrightarrow} \hat{c}'.\mathit{rgn}' \wedge \hat{c}.\mathit{rgn} \leq^{\mathrm{r}} \hat{c}'.\mathit{rgn}'$$

$$\textbf{WFRegions}(P) \equiv\ \leq^{\mathrm{r}} \text{ is antisymmetric}$$

$$\textbf{CompleteRegions}(P) \equiv \mathrm{rng}(\prec^{\mathrm{r}}) \subseteq \mathrm{dom}(\prec^{\mathrm{r}}) \cup \{\textsf{Object.Instance}\}$$

$$\textbf{NoShadowing}(P) \equiv \ldots\ \wedge\ (c.\mathit{rgn} \Subset^{\mathrm{c}} c \Rightarrow (\nexists c' : c \neq c' \wedge c \leq^{\mathrm{c}} c' \wedge c'.\mathit{rgn} \Subset^{\mathrm{c}} c'))$$

$$\begin{aligned}
&\textbf{AggregationsOK}(P) \equiv \forall c, c', \mathit{rgn}, \mathit{argn} : \\
&\qquad \textsf{class}\ c' \cdots \{ \cdots \textsf{unshared}\ [\textsf{final}]_{\mathrm{opt}}\ c \cdots \textsf{aggregate}\ \cdots \mathit{rgn}\ \textsf{into}\ \mathit{argn} \cdots = \cdots \} \text{ is in } P \\
&\qquad \Rightarrow \mathit{rgn} \overset{c}{\hookrightarrow} \hat{c}.\mathit{rgn} \wedge \mathit{argn} \overset{c'}{\hookrightarrow} \hat{c}'.\mathit{argn}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{M}^{c}_{\mathit{fd}}(\hat{c}.\mathit{rgn}) = \hat{c}'.\mathit{argn} \Leftrightarrow \\
&\qquad \textsf{class}\ c' \cdots \{ \cdots \textsf{unshared}\ [\textsf{final}]_{\mathrm{opt}}\ c\ \mathit{fd} \cdots \textsf{aggregate}\ \cdots \mathit{rgn}\ \textsf{into}\ \mathit{argn} \cdots = \cdots \} \text{ is in } P \\
&\qquad \wedge\ \mathit{rgn} \overset{c}{\hookrightarrow} \hat{c}.\mathit{rgn}\ \wedge\ \mathit{argn} \overset{c'}{\hookrightarrow} \hat{c}'.\mathit{argn}
\end{aligned}$$

$$\begin{aligned}
&\textbf{MapOK}(P, \mathcal{M}, c) \equiv \\
&\qquad (\forall (c_1.\mathit{rgn}, c_2.\mathit{argn}) \in \mathcal{M},\ (c_3.\mathit{rgn}', c_4.\mathit{argn}') \in \mathcal{M} : c_1.\mathit{rgn} \leq^{\mathrm{r}} c_3.\mathit{rgn}' \Rightarrow c_2.\mathit{argn} \leq^{\mathrm{r}} c_4.\mathit{argn}') \\
&\qquad \wedge\ (\forall c'.\mathit{rgn}' \in^{\mathrm{c}} c : \exists \hat{c}.\hat{\mathit{rgn}} \in \mathrm{dom}(\mathcal{M}) : c'.\mathit{rgn}' \leq^{\mathrm{r}} \hat{c}.\hat{\mathit{rgn}})
\end{aligned}$$

$$\begin{aligned}
&\langle \mathit{mn}, t_1 \ldots t_n \xrightarrow{\textsf{reads}\ R\ \textsf{writes}\ W} t, (x_1 \ldots x_n), e \rangle \Subset^{\mathrm{c}} c \\
&\qquad \Leftrightarrow \\
&\textsf{class}\ c \cdots \{ \cdots t\ \mathit{mn}_{\mathit{lbl}}([\textsf{final}]_{\mathrm{opt}}\ t_1\ [x_1]_{\mathit{lbl}_1} \ldots [\textsf{final}]_{\mathrm{opt}}\ t_n\ [x_n]_{\mathit{lbl}_n})\ \textsf{reads}\ R\ \textsf{writes}\ W\ \{e\} \cdots \} \text{ is in } P
\end{aligned}$$

$$\begin{aligned}
&\textbf{ClassMethodsOK}(P) \equiv \forall c, c', e, e', \mathit{mn}, t_i, t_i', t, t', V, V', \psi, \psi' : \\
&\qquad (\langle \mathit{mn}, t_1 \ldots t_n \xrightarrow{\psi} t, V, e \rangle \Subset^{\mathrm{c}} c\ \wedge\ \langle \mathit{mn}, t'_1 \ldots t'_n \xrightarrow{\psi'} t', V', e' \rangle \Subset^{\mathrm{c}} c') \\
&\qquad \Rightarrow ((t = t'\ \wedge\ t_i = t_i'\ \wedge\ V = V'\ \wedge\ \psi = \psi')\ \vee\ (c \not\leq^{\mathrm{c}} c'))
\end{aligned}$$

$$c \overset{\mathrm{fx}}{\hookrightarrow} \textsf{reads}\ R\ \textsf{writes}\ W \Leftrightarrow \textsf{class}\ c \cdots \textsf{reads}\ R\ \textsf{writes}\ W\ \{\cdots\} \text{ is in } P$$

Figure 4.7: Additional predicates and relations for checking FLUIDJAVA extended with regions and effects. Metavariables R and W are lists $(tgt_1 \ldots tgt_n)$.

- We treat an effect reads $tgt_1^R, \ldots, tgt_n^R$ writes $tgt_1^W, \ldots, tgt_m^W$ as the set $\bigcup_{i=1}^{n}\{\textsf{reads}\ tgt_i^R\} \cup \bigcup_{j=1}^{m}\{\textsf{writes}\ tgt_j^W\}$, and vice versa.
- If $\psi \in \mathit{effect}$ then $\psi[e/v]$, where v is a variable name or this and e is an expression, is the set of effects where targets of the form $v.\mathit{rgn}$ are replaced with the target $e.\mathit{rgn}$. In particular, substitution *does not* occur for v's that appear within an expression e.

Additional predicates and relations necessary for insuring the region hierarchy is well formed are shown in Figure 4.7. Recall that the set of region names is the union of the set of abstract region names and the set of field names. We thus replace the predicate $\textbf{FieldsOnce}(P)$ with the predicate $\textbf{RegionsOnce}(P)$ to enforce that region names are unique within a class. To check that regions form a hierarchy we construct an order over the regions based on the declarations. (1) We extend

$\Subset^{\mathrm{c}}$ and $\in^{\mathrm{c}}$ to relate abstract regions to classes. (2) We assume that $\mathsf{Object.Instance} \Subset^{\mathrm{c}} \mathsf{Object}$. (3) We define new relations $\prec^{\mathrm{r}}$ and $\leq^{\mathrm{r}}$ over all regions; $c.rgn \prec^{\mathrm{r}} c'.rgn'$ only when rgn', the intended parent region of rgn, is defined in class c. Predicate $\mathbf{WFRegions}(P)$ checks that the regions form a hierarchy. The triple $rgn \overset{c}{\mapsto} c'.rgn$ looks up the particular region with name rgn that is defined in class c, and gives its fully qualified name. Similarly, $rgn \leq^{\mathrm{r}}_{c,c'} rgn'$ compares two regions based on the results of looking up the region names in classes c and c'. To check that the regions used as superregions are defined we define $\mathbf{CompleteRegions}(P)$. We update $\mathbf{NoShadowing}(P)$ to additionally prevent shadowing of abstract regions.

For an $\mathsf{unshared}$ field fd in class c that references an object of class c', $\mathbf{AggregationsOK}(P)$ checks that the associated region mapping maps regions defined in c' to abstract regions defined in class c. Each $\mathsf{unshared}$ field defines a function $\mathcal{M}^{c}_{fd}$ that represents its aggregation mapping. The predicate $\mathbf{MapOK}(P, \mathcal{M}, c)$ insures that this mapping respects the class hierarchy and that every region of the uniquely referenced object is mapped (perhaps indirectly) into a region of the referencing object. The relations $\Subset^{\mathrm{c}}$ and $\in^{\mathrm{c}}$ as applied to field declarations are updated in the obvious and tedious manner (not shown).

We extend the type signature of a method in the relation $\Subset^{\mathrm{c}}$ to include its declared effects; the relation $\in^{\mathrm{c}}$, not shown, is also updated in the obvious manner. Predicate $\mathbf{MethodsOnce}(P)$, also not shown, is updated to handle the extended method signature as well. We also extend $\mathbf{ClassMethodsOK}(P)$ so that method overriding preserves declared effects. Parameter names can appear in declared effects; we must be careful, therefore, in how we compare the declared effects. In the interest of simplicity, we deviate from the original definition of the predicate and require that method overriding preserve parameter names so that the declared effects may be directly compared. For simplicity, we also require that overriding methods declare exactly the same effects. An alternative is to allow the declaration of more specific effects, in which case we would have to check that all the effects of the overriding declaration are accounted for in the original declaration.

The new relation $c \overset{\mathrm{fx}}{\mapsto} \psi$ maps a class name to its constructor effects. This is necessary because FLUIDJAVA does not have formal constructors; if it did we would treat them similarly to methods. We assume that $\mathsf{Object} \overset{\mathrm{fx}}{\mapsto} \mathsf{reads\ nothing\ writes\ nothing}$.

For the full Java language, we also have to check that no static region is a subregion of an instance region, and that no region is more visible than its parent region.

Judgment	*Meaning*
$\vdash P : t\,!\,\psi$	Program P yields type t and has effect ψ
$P \vdash \mathit{defn}$	defn is a well formed definition
$P; E \vdash_{\mathrm{decl}} \mathit{effect}$	effect is a well formed effect declaration
$P; E \vdash_{\mathrm{decl}} \mathit{tgt}$	tgt is a well formed target within an effect declaration
$P; E \vdash \mathit{effect}$	effect is a well formed effect
$P; E \vdash \mathit{tgt}$	tgt is a well formed target
$P \vdash t$	t exists
$P, c \vdash \mathit{field}\,!\,\psi$	field is a well formed field whose initializer has effect ψ
$P, c \vdash \mathit{meth}$	meth is a well formed method in type c
$P; E \vdash e : t\,!\,\psi$	Expression e has type t and effect ψ

Figure 4.8: Typing judgments for the extended language.

$$\frac{P \vdash \mathit{tgt}_i \rightsquigarrow r}{P \vdash \textsf{reads}\ \mathit{tgt}_1 \ldots \mathit{tgt}_j\ \textsf{writes}\ \mathit{tgt}_{j+1} \ldots \mathit{tgt}_n} \qquad \frac{}{P \vdash \textsf{nothing} \rightsquigarrow \cdot} \qquad \frac{\mathcal{E}_P(\mathit{lbl}) = E_1, [\textsf{final}]_{\mathrm{opt}}\ c\ x, E_2}{P \vdash [x]_{\mathit{lbl}} \rightsquigarrow \cdot}$$

$$\frac{P; \mathcal{E}_P(\mathit{lbl}) \vdash e : c\,!\,\psi \qquad c'.\mathit{rgn} \in^{\mathrm{c}} c}{P \vdash [e]_{\mathit{lbl}}.\mathit{rgn} \rightsquigarrow c'.\mathit{rgn}} \qquad \frac{c'.\mathit{rgn} \in^{\mathrm{c}} c}{P \vdash \textsf{any}(c).\mathit{rgn} \rightsquigarrow c'.\mathit{rgn}}$$

Figure 4.9: Rules for well formed effects and targets.

4.11.2 Typing Rules

Typing judgments must incorporate effects, and now have the forms shown in Figure 4.8. We now have rules for well formed effects and targets. There are two sets of rules, one that limits the shape of a target within an effects declaration clause, and a second set for general effects and targets. Figure 4.9 shows the general rules. An effect is well formed if all its targets are well formed. A target is well formed if (1) it is the empty target; (2) it is a local target and the variable is defined in the environment; (3) it is an instance target and the region is defined in the type of the expression; or (4) it is an any-instance target and the region is defined in the named type. Environments are determined from the label of the expression contained within the target. Well formed targets, therefore, must contain labeled expressions. The judgments for well formed instance and any-instance targets also elaborate to the fully qualified name of the region that is named in the target.

The rules for well formed effects declarations and well formed targets within effect declarations are shown in figure 4.10. A target in an effects declaration must be the empty target, an instance target whose expression is a method parameter, including this, or an any-instance target. The limi-

$$\frac{P; E \vdash_{\text{decl}} tgt_i}{P; E \vdash_{\text{decl}} \mathsf{reads}\ tgt_1 \ldots tgt_j\ \mathsf{writes}\ tgt_{j+1} \ldots tgt_n} \qquad \frac{}{P; E \vdash_{\text{decl}} \mathsf{nothing}}$$

$$\frac{E = E_1, [\mathsf{final}]_{\text{opt}}\ c\ x, E_2 \qquad c'.rgn \in^{\text{c}} c}{P; E \vdash_{\text{decl}} x.rgn} \qquad \frac{c'.rgn \in^{\text{c}} c}{P; E \vdash_{\text{decl}} \mathsf{any}(c).rgn}$$

Figure 4.10: Rules for well formed effect declarations.

NULL
$$\frac{P \vdash c}{P; E \vdash \mathsf{null} : c\,!\,\emptyset}$$

ABSTRACT
$$\frac{P \vdash t}{P; E \vdash \mathsf{abstract} : t\,!\,\emptyset}$$

FINALVAR
$$\frac{E = E_1, \mathsf{final}\ t\ x, E_2}{P; E \vdash x : t\,!\,\emptyset}$$

Figure 4.11: Rules for expressions without effects.

tation to method parameters is enforced by checking the well formedness of the effects declaration in an environment that only contains method parameters; see rule METHOD in Figure 4.15. Rule CLASS checks the effects of the constructor using an environment containing this only.

We discuss the program type and effects rules bottom up, starting with effect-producing expressions, and moving upward along the syntax to method and class declarations. Figure 4.11 gives the rules for those expressions that do not produce or propagate any effects. Evaluating the null expression does not produce any effects, nor does evaluating any other primitive integer or boolean literal. The abstract expression has no effect—it is treated like an empty method body. Reading a final local variable produces no effect because such a variable does not represent mutable state.

The rules for expressions that have direct effects are shown in Figure 4.12. These are the expressions that *cause* effects, as opposed to merely propagating the effects of other expressions. Reading from and writing to a non-final local variable produce read and write effects, respectively. Reading from a non-final field causes a read effect to the field and propagates the effects of evaluating the

VAR
$$\frac{E = E_1, t\ x, E_2}{P; E \vdash [x]_{lbl} : t\,!\,\{\text{reads}\ [x]_{lbl}\}}$$

ASSIGN
$$\frac{P; E \vdash e : t\,!\,\psi \qquad E = E_1, t\ x, E_2}{P; E \vdash [x]_{lbl} = e : t\,!\,\psi \cup \{\text{writes}\ [x]_{lbl}\}}$$

GET
$$\frac{P; E \vdash [e]_{lbl} : c\,!\,\psi \qquad \langle c'.fd, t\rangle \in^{\text{c}} c}{P; E \vdash [e]_{lbl}.fd : t\,!\,\psi \cup \{\text{reads}\ [e]_{lbl}.fd\}}$$

SET
$$\frac{P; E \vdash [e]_{lbl} : c\,!\,\psi_1 \qquad \langle c'.fd, t\rangle \in^{\text{c}} c \qquad P; E \vdash e' : t\,!\,\psi_2}{P; E \vdash [e]_{lbl}.fd = e' : t\,!\,\psi_1 \cup \psi_2 \cup \{\text{writes}\ [e]_{lbl}.fd\}}$$

Figure 4.12: Rules for expressions with direct effects.

$$\frac{P \vdash c \qquad c \overset{\text{fx}}{\mapsto} \psi \qquad \textbf{NoAbstractMethods}(P, c)}{P; E \vdash \mathsf{new}\ c : c\,!\,\psi}\ \textsc{New}$$

$$\frac{P; E \vdash e : c\,!\,\psi_{\mathrm{e}} \qquad P; E \vdash e_i : t_i\,!\,\psi_i \qquad \langle mn, t_1 \ldots t_n \xrightarrow{\psi} t, (x_1 \ldots x_n), e_b\rangle \in^{\mathrm{c}} c}{P; E \vdash e.mn(e_1 \ldots e_n) : t\,!\,\psi_{\mathrm{e}} \cup (\bigcup_{i=1}^{n} \psi_i) \cup \psi[e/\mathsf{this}][e_i/x_i]}\ \textsc{Invoke}$$

$$\frac{P; E \vdash \mathsf{this} : c'\,!\,\psi_{\mathrm{R}} \qquad c' \prec^{\mathrm{c}} c \qquad \langle mn, t_1 \ldots t_n \xrightarrow{\psi} t, (v_1 \ldots v_n), e_b\rangle \in^{\mathrm{c}} c \qquad e_b \neq \mathsf{abstract} \qquad P; E \vdash e_i : t_i\,!\,\psi_i}{P; E \vdash \mathsf{super}.mn(e_1 \ldots e_n) : t\,!\,\psi_{\mathrm{R}} \cup (\bigcup_{i=1}^{n} \psi_i) \cup \psi[e_i/v_i]}\ \textsc{Super}$$

Figure 4.13: Rules for object creation and method calls.

expression describing the receiver. Similarly for assigning to a field.

The rules for object creation and method calls are shown in Figure 4.13. Constructing an object has the effects declared on the class, obtained using the $\overset{\text{fx}}{\mapsto}$ relation. If we had named constructors in our language, then the rule would be similar to the method invocation rule. Invoking a method causes the latent effects ψ of the method, and also propagates the effects of evaluating the expressions for the receiver, ψ_{e} and the method arguments, ψ_i. Any parameter names, including the receiver, in the targets of the method's effects are simultaneously replaced by the actual expressions. For example, if method `Point.add(Point p)` has the effects

$$\{\mathsf{reads\ this.}\texttt{Location}, \mathsf{reads\ p.}\texttt{Location}\}$$

then the expression `pt1.add(pt2)`, has the effects

$$\mathsf{reads}\ \texttt{pt1}, \texttt{pt2}, \texttt{pt1}.\texttt{Location}, \texttt{pt2}.\texttt{Location}\ \mathsf{writes\ nothing}$$

The rule for a invoking a superclass implementation of a method is similar, except that there is no substitution for this. The receiver effect ψ_{R} is always {reads this} because it originates from a judgment using the local variable this.

Rules for the rest of the expressions are shown in Figure 4.14. The effects of all these expressions are simply the union of the effects of their subexpressions. Reading from a final field does not produce an effect on that field because it is not considered to be mutable state. Subsumption

$$\textsc{FinalGet}\quad \frac{P; E \vdash e : c \,!\, \psi \qquad \langle c'.\mathit{fd}, \mathsf{final}\ t\rangle \in^{\mathrm{c}} c}{P; E \vdash e.\mathit{fd} : t \,!\, \psi}$$

$$\textsc{If}\quad \frac{P; E \vdash e_1 : \mathsf{boolean} \,!\, \psi_1 \qquad P; E \vdash e_2 : t \,!\, \psi_2 \qquad P; E \vdash e_3 : t \,!\, \psi_3}{P; E \vdash \mathsf{if}(e_1)\ \{e_2\}\ \mathsf{else}\ \{e_3\} : t \,!\, \psi_1 \cup \psi_2 \cup \psi_3}$$

$$\textsc{Seq}\quad \frac{P; E \vdash e_1 : t_1 \,!\, \psi_1 \qquad P; E \vdash e_2 : t_2 \,!\, \psi_2}{P; E \vdash e_1; e_2 : t_2 \,!\, \psi_1 \cup \psi_2}$$

$$\textsc{Let}\quad \frac{\mathit{mod\_t} = [\mathsf{final}]_{\mathrm{opt}}\ t \qquad x \notin E \qquad P; E \vdash e_1 : t \,!\, \psi_1 \qquad P; E, \mathit{mod\_t}\ x \vdash e_2 : t' \,!\, \psi_2}{P; E \vdash \mathsf{let}\ \mathit{mod\_t}\ x = e_1\ \mathsf{in}\ \{e_2\} : t' \,!\, \psi_1 \cup \psi_2}$$

$$\textsc{Sub}\quad \frac{P; E \vdash e : c' \,!\, \psi \qquad c' \leq^{\mathrm{c}} c}{P; E \vdash e : c \,!\, \psi}$$

$$\textsc{Sync}\quad \frac{P; E \vdash e_1 : c \,!\, \psi_1 \qquad P; E \vdash e_2 : t \,!\, \psi_2}{P; E \vdash \mathsf{synchronized}(e_1)\ \{e_2\} : t \,!\, \psi_1 \cup \psi_2}$$

$$\textsc{Fork}\quad \frac{P; E \vdash e : t \,!\, \psi}{P; E \vdash \mathsf{fork}\ \{e\} : \mathsf{int} \,!\, \psi}$$

$$\textsc{Label}\quad \frac{P; E \vdash e : t \,!\, \psi \qquad \mathcal{E}_P \supseteq \{(\mathit{lbl}, E)\}}{P; E \vdash [e]_{\mathit{lbl}} : t \,!\, \psi}$$

Figure 4.14: Rules for expressions with only indirect effects.

is not allowed to widen the effects of the expression. Fork is interesting because it does not have an equivalent expression in Java, rather it corresponds to an invocation of the `Thread.start()` method. The "body" for the thread is defined by an implementation of the `Runnable` interface's `run()` method. In general, this means the effect of starting a thread in Java is going to be `@writes Object.All`.

Finally, the rules for well formed definitions and programs are in Figure 4.15. The predicate antecedents in rule PROG are updated as previously described. There is an additional rule for unshared fields: it includes the previously described **MapOK** predicate as an antecedent. The rules for fields propagate the effect of field initialization expressions. These effects are gathered in the CLASS rule and checked against the declared effects of the class. The declared effects of a class must also include the effects of constructing an instance of its superclass. The effects of the body of a method are checked against the declared effects of the method. Predicate $\mathbf{CheckFX}(P,\ \mathit{args},\ \psi,\ \psi')$, where args is a map from parameter names to their declared types, is true when "all the effects of ψ are included in the effects of ψ'."

4.11.3 Checking Declared Effects

We now return to the problem of checking the declared effects of a method against the actual effects of the method's implementation. This checking is abstractly described in Section 4.3.3. We use an "elaboration" that expands a set of effects to take into account the bindings of local variables and the mapping of unshared fields. This process is defined in Figure 4.16. For each target *tgt*

PROG
$$\frac{\begin{array}{c}\mathbf{ClassOnce}(P) \quad \mathbf{RegionsOnce}(P) \quad \mathbf{MethodsOnce}(P) \quad \mathbf{NoShadowing}(P) \quad \mathbf{CompleteClasses}(P) \\ \mathbf{WFClasses}(P) \quad \mathbf{ClassMethodsOK}(P) \quad \mathbf{UniqueLabels}(P) \quad \mathbf{WFRegions}(P) \\ \mathbf{CompleteRegions}(P) \quad \mathbf{AggregationsOK}(P) \quad P = \mathit{defn}_1 \ldots \mathit{defn}_n\ e \quad P \vdash \mathit{defn}_i \quad P;\emptyset \vdash e : t\,!\,\psi\end{array}}{\vdash P : t\,!\,\psi}$$

CLASS
$$\frac{\begin{array}{c}\mathbf{CheckFX}(P,\ \emptyset,\ \mathbf{mask}(P,\ \mathbf{elaborate}(P,\ \psi \cup \bigcup_{i=1}^{j} \psi_i),\ \emptyset),\ \mathit{effect}) \\ P; \mathsf{final}\ \mathit{cn}\ \mathsf{this} \vdash_{\text{decl}} \mathit{effect} \quad P, \mathit{cn} \vdash \mathit{field}_i\,!\,\psi_i \quad P, \mathit{cn} \vdash \mathit{meth}_i \quad c' \overset{\text{fx}}{\mapsto} \psi\end{array}}{P \vdash \mathsf{class}\ \mathit{cn}\ \mathsf{extends}\ c' \cdots \mathit{effect}\ \{\cdots \mathit{field}_1 \ldots \mathit{field}_j\ \mathit{meth}_1 \ldots \mathit{meth}_k\}}$$

FIELD
$$\frac{P \vdash t \qquad P;\emptyset \vdash e : t\,!\,\psi}{P, c \vdash [\mathsf{final}]_{\text{opt}}\ t\ \mathit{fd}\ \mathsf{in}\ \mathit{rgn} = e\,!\,\psi}$$

UNSHAREDFIELD
$$\frac{P \vdash c' \qquad P;\emptyset \vdash e : c'\,!\,\psi \qquad \mathbf{MapOK}(P,\ \mathcal{M}^{c}_{\mathit{fd}},\ c')}{P, c \vdash \mathsf{unshared}\ [\mathsf{final}]_{\text{opt}}\ c'\ \mathit{fd}\ \mathsf{in}\ \mathit{rgn}\ \mathsf{aggregate}\ \mathit{agg}_1 \ldots \mathit{agg}_n\ = e\,!\,\psi}$$

METHOD
$$\frac{\begin{array}{c}P \vdash t \quad P \vdash \mathit{mod\_t}_i \quad E = \mathsf{final}\ c\ \mathsf{this}, \mathit{mod\_t}_1\ x_1, \ldots, \mathit{mod\_t}_n\ x_n \quad P;E \vdash e : t\,!\,\psi \quad P;E \vdash_{\text{decl}} \mathit{effect} \\ \mathit{mod\_t}_i = [\mathsf{final}]_{\text{opt}}\ t_i \quad \mathbf{CheckFX}(P,\ \{\mathsf{this} \mapsto c,\ x_i \mapsto t_i\},\ \mathbf{mask}(P,\ \mathbf{elaborate}(P,\ \psi),\ \{\mathsf{this},\ x_i\}),\ \mathit{effect}) \\ \mathcal{E}_P \supseteq \{(\mathit{lbl},\ E), (\mathit{lbl}_1,\ E), \ldots, (\mathit{lbl}_n,\ E)\}\end{array}}{P, c \vdash t\ \mathit{mn}_{\mathit{lbl}}(\mathit{mod\_t}_1\ [x_1]_{\mathit{lbl}_1} \ldots \mathit{mod\_t}_n\ [x_n]_{\mathit{lbl}_n})\ \mathit{effect}\ \{e\}}$$

Figure 4.15: Rules for well formed definitions with regions and effects. **elaborate**, **mask**, and **CheckFX** are defined in Figures 4.16, 4.17, and 4.18, respectively.

that describes a region of an object referenced by a local variable, (1) binding context analysis is used to find the possible source expressions of the reference and (2) new targets—that describe the same state as *tgt*—based on these expressions are added to the set of targets. The purpose of this is to avoid naming state through local variables which are meaningless outside of the method body. In Java, Binding Context Analysis traces the source of a local's value to (1) a method parameter, (2) a `new` expression, (3) a method call, or (4) a field reference, including an array element. As previously discussed, and as formalized below, effects on newly created objects are ignored. An instance target of the form $[x]_{\mathit{lbl}}.\mathit{rgn}$ where x binds to method parameters and `new` expressions only can be accounted for by declared effects on regions of those parameters. This is our implementation of the covers relation of Section 4.3.3.

Elaboration also exploits the mapping of unshared fields. In general, ignoring labels, targets of the form $e.\mathit{fd}.\mathit{rgn}$ are replaced by targets of the form $e.\mathit{rgn}'$ when fd is an unshared field and rgn is a subregion of a region that maps to rgn'. When the region mapping is ambiguous, because two related regions are both explicitly mapped, we always map into the region closest to the leaves of the region hierarchy. In combination with Binding Context Analysis expanding references through locals into references through parameters and fields, this process effectively chases a chain of unique object

$$\mathbf{map}(P,\, c.rgn,\, \mathcal{M}) \equiv \min_{\leq^{\mathrm{r}}} \{\hat{c}.\hat{rgn} \mid c.rgn \leq^{\mathrm{r}} c'.rgn' \wedge \mathcal{M}(c'.rgn') = \hat{c}.\hat{rgn}\}$$

$$\mathbf{elaborate}(P,\, \textsf{reads } R \textsf{ writes } W) \equiv \textsf{reads } \mathbf{elaborate}(P,\, R) \textsf{ writes } \mathbf{elaborate}(P,\, W)$$

$$\mathbf{elaborate}(P,\, T') \equiv \text{The smallest set } T \supseteq T' \text{ such that}$$
$$\left\{ \begin{array}{l} [x]_{lbl}.rgn \in T \Rightarrow T \supseteq \{e.rgn \mid e \in B\ [x]_{lbl}\ \} \\ [[e]_{lbl}.fd]_{lbl'}.rgn' \in T \Rightarrow T \supseteq \left\{ [e]_{lbl}.rgn \;\middle|\; \begin{array}{l} P; \mathcal{E}_P(lbl) \vdash e : c\,!\,\psi \\ \wedge\ P; \mathcal{E}_P(lbl') \vdash [e]_{lbl}.fd : c'\,!\,\psi' \\ \wedge\ rgn' \overset{c'}{\mapsto} \hat{c}'.rgn'\ \wedge\ \hat{c}.rgn = \mathbf{map}(P,\, \hat{c}'.rgn,\, \mathcal{M}^{c}_{fd}) \end{array} \right\} \end{array} \right.$$

Figure 4.16: Effect elaboration.

$$\mathbf{unshared}(P,\, [e]_{lbl}.fd) \equiv$$
$$(P; \mathcal{E}_P(lbl) \vdash e : c'\,!\,\psi)\ \wedge\ fd \overset{c'}{\mapsto} c.fd\ \wedge\ \exists \hat{c} : \textsf{class } c \cdots \{\cdots \textsf{unshared } [\textsf{final}]_{\mathrm{opt}}\ \hat{c}\ fd \cdots\} \text{ is in } P$$

$$\mathbf{mask}(P,\, \textsf{reads } R \textsf{ writes } W,\, args) \equiv \textsf{reads } \mathbf{mask}(P,\, R,\, args) \textsf{ writes } \mathbf{mask}(P,\, W,\, args)$$

$$\mathbf{mask}(P,\, T,\, args) \equiv \{tgt \in T \mid \mathbf{mask}(P,\, tgt,\, args)\}$$

$$\mathbf{mask}(P,\, tgt,\, args) \equiv \begin{cases} \text{true} & (tgt \equiv [x]_{lbl}) \\ x \notin args & (tgt \equiv [x]_{lbl}.rgn) \\ \text{true} & (tgt \equiv [\textsf{new } c]_{lbl}.rgn) \\ \mathbf{unshared}(P,\, e.fd) & (tgt \equiv [e.fd]_{lbl}.rgn) \\ \text{false} & (\text{Otherwise}) \end{cases}$$

Figure 4.17: Effect masking.

references to the "largest" aggregation visible within the method body. Ideally this aggregation would be via an object identifiable through a method parameter (usually `this`), which would allow all the effects on the aggregation to be accounted for by a declared effect on a target of the form $x.rgn$.

Elaboration is followed by effect masking, defined in Figure 4.17, which in addition to removing effects based on the criteria described in Section 4.3.3, also removes those effects rendered redundant because of elaboration.[8] More specifically, for FLUIDJAVA,

- We mask effects on local targets.
- We mask effects on instance targets $x.rgn$ where x is not a method parameter. Such effects are redundant after elaboration and to keep them would require the declared effects to use any-instance targets.
- We mask effects on newly created objects.
- We mask effects on unshared objects. Such effects are redundant after elaboration, which replaces them with effects on an aggregated object.

[8]We cannot remove the redundant effects during elaboration because otherwise it is not possible to reach a fixed point.

$\textbf{CheckFX}(P,\ \mathit{args},\ \textsf{reads}\ R\ \textsf{writes}\ W,\ \textsf{reads}\ R'\ \textsf{writes}\ W')$ $\equiv$ $(\forall \mathit{tgt} \in R : \textbf{CheckTgt}(P,\ \mathit{args},\ \mathit{tgt},\ R' \cup W'))\ \wedge\ (\forall \mathit{tgt} \in W : \textbf{CheckTgt}(P,\ \mathit{args},\ \mathit{tgt},\ W'))$
$\textbf{CheckTgt}(P,\ \mathit{args},\ \textsf{any}(c).\mathit{rgn},\ T) \equiv \exists \textsf{any}(c').\mathit{rgn}' \in T : c \leq^{\mathrm{c}} c' \wedge \mathit{rgn} \leq^{\mathrm{r}}_{c,c'} \mathit{rgn}'$
$\textbf{CheckTgt}(P,\ \mathit{args},\ x.\mathit{rgn},\ T) \equiv$ $c = \mathit{args}(x) \wedge \big(\big(\exists x.\mathit{rgn}' \in T : \mathit{rgn} \leq^{\mathrm{r}}_{c,c} \mathit{rgn}' \big) \vee \big(\exists \textsf{any}(c').\mathit{rgn}' \in T : c \leq^{\mathrm{c}} c' \wedge \mathit{rgn} \leq^{\mathrm{r}}_{c,c'} \mathit{rgn}' \big) \big)$
$\textbf{CheckTgt}(P,\ \mathit{args},\ [e]_{\mathit{lbl}}.\mathit{rgn},\ T) \equiv$ $\exists \textsf{any}(c').\mathit{rgn}' \in T : (P; \mathcal{E}_P(\mathit{lbl}) \vdash [e]_{\mathit{lbl}} : c \,!\, \psi) \wedge c \leq^{\mathrm{c}} c' \wedge \mathit{rgn} \leq^{\mathrm{r}}_{c,c'} \mathit{rgn}'$

Figure 4.18: Predicates for checking implementation effects against declared effects.

The implementation effects of a method are checked against the method's declared effects by first elaborating them and masking the results. Each remaining read effect must be accounted for by a declared read or write effect; each remaining write effect must be accounted for a declared write effect. An affected any-instance target is accounted for by an any-instance target identifying a superregion of a superclass. An instance target based on a method parameter can be accounted for by a target identifying a superregion of the same parameter. Otherwise, in general, an instance target is accounted for by an any-instance target identifying a superregion of a superclass.

4.11.4 Overlap and Conflict

We conclude by describing how to compare the effects of two expressions to determine if the effects conflict. That is, if one of the expressions writes something that the other one reads or writes. We start by defining *target overlap* in Figure 4.19: we say two targets overlap if they identify state that *may* intersect. Two local targets overlap if they refer to the same local variable. Two non-local targets overlap only if they can refer to overlapping regions of the same object, where two regions overlap if one is a subregion of the other. Two instance targets can overlap only if the objects they refer to could be identical. To compare instance targets, we must determine whether two expressions from different points in the program, may refer to overlapping sets of locations. This observation has led us to formalize the desired notion of equality, based on the context of the expressions, as a new alias question *MayEqual*$([e]_{lbl}, [e']_{lbl})$ [BG99]. More traditional alias analyses, such as Steensgaard's "points-to" analysis [Ste96], may be used as a conservative approximation.

Two well formed instance targets overlap if they may refer to overlapping regions of the the same object. A well formed any-instance target may refer to the same object as a well formed instance target if the type of the object that the instance target refers to has a subtype that is also a subtype of the type declared in the any-instance target. That is, it is possible that the object referred

$$c.\mathit{rgn}\ \mathsf{overlap}^{\mathrm{r}}\ c'.\mathit{rgn}' \quad\equiv\quad c.\mathit{rgn} \leq^{\mathrm{r}} c'.\mathit{rgn}' \ \vee\ c'.\mathit{rgn}' \leq^{\mathrm{r}} c.\mathit{rgn}$$

$$\frac{P \vdash \mathit{tgt}'\ \mathsf{overlap}^{\mathrm{t}}\ \mathit{tgt}}{P \vdash \mathit{tgt}\ \mathsf{overlap}^{\mathrm{t}}\ \mathit{tgt}'} \qquad \frac{P \vdash [x]_{\mathit{lbl}} \qquad P \vdash [x']_{\mathit{lbl}'} \qquad x = x'}{P \vdash [x]_{\mathit{lbl}}\ \mathsf{overlap}^{\mathrm{t}}\ [x']_{\mathit{lbl}'}}$$

$$\frac{P \vdash [e]_{\mathit{lbl}}.\mathit{rgn} \rightsquigarrow \hat{c}.\mathit{rgn} \qquad P \vdash [e']_{\mathit{lbl}'}.\mathit{rgn} \rightsquigarrow \hat{c}'.\mathit{rgn}' \qquad \mathit{MayEqual}([e]_{\mathit{lbl}}, [e']_{\mathit{lbl}'}) \qquad \hat{c}.\mathit{rgn}\ \mathsf{overlap}^{\mathrm{r}}\ \hat{c}'.\mathit{rgn}'}{P \vdash [e]_{\mathit{lbl}}.\mathit{rgn}\ \mathsf{overlap}^{\mathrm{t}}\ [e']_{\mathit{lbl}'}.\mathit{rgn}'}$$

$$\frac{P \vdash [e]_{\mathit{lbl}}.\mathit{rgn} \rightsquigarrow \hat{c}.\mathit{rgn} \qquad P \vdash \mathsf{any}(c').\mathit{rgn}' \rightsquigarrow \hat{c}'.\mathit{rgn}' \qquad \exists t' : t' \leq^{\mathrm{c}} c \wedge t' \leq^{\mathrm{c}} c' \qquad \hat{c}.\mathit{rgn}\ \mathsf{overlap}^{\mathrm{r}}\ \hat{c}'.\mathit{rgn}'}{P \vdash [e]_{\mathit{lbl}}.\mathit{rgn}\ \mathsf{overlap}^{\mathrm{t}}\ \mathsf{any}(c').\mathit{rgn}'}$$

$$\frac{P \vdash \mathsf{any}(c).\mathit{rgn} \rightsquigarrow \hat{c}.\mathit{rgn} \qquad P \vdash \mathsf{any}(c').\mathit{rgn}' \rightsquigarrow \hat{c}'.\mathit{rgn}' \qquad \exists t' : t' \leq^{\mathrm{c}} c \wedge t' \leq^{\mathrm{c}} c' \qquad \hat{c}.\mathit{rgn}\ \mathsf{overlap}^{\mathrm{r}}\ \hat{c}'.\mathit{rgn}'}{P \vdash \mathsf{any}(c).\mathit{rgn}\ \mathsf{overlap}^{\mathrm{t}}\ \mathsf{any}(c').\mathit{rgn}'}$$

Figure 4.19: Rules for region and target overlap.

$$\frac{\exists t \in T, t' \in T' : (P \vdash t\ \mathsf{overlap}^{\mathrm{t}}\ t')}{P \vdash T\ \mathsf{overlap}^{\mathrm{t}}\ T'} \qquad \frac{P \vdash \mathbf{elaborate}(P,\ R \cup W)\ \mathsf{overlap}^{\mathrm{t}}\ \mathbf{elaborate}(P,\ W')}{P \vdash (\mathsf{reads}\ R\ \mathsf{writes}\ W)\ \mathsf{conflict}\ (\mathsf{reads}\ R'\ \mathsf{writes}\ W')}$$

$$\frac{P \vdash \psi'\ \mathsf{conflict}\ \psi}{P \vdash \psi\ \mathsf{conflict}\ \psi'} \qquad \frac{P; \mathcal{E}_P(\mathit{lbl}) \vdash [e]_{\mathit{lbl}} : t \,!\, \psi \qquad P; \mathcal{E}_P(\mathit{lbl}') \vdash [e']_{\mathit{lbl}'} : t' \,!\, \psi' \qquad P \vdash \psi\ \mathsf{conflict}\ \psi'}{P \vdash [e]_{\mathit{lbl}}\ \mathsf{interfere}\ [e']_{\mathit{lbl}'}}$$

Figure 4.20: Rules for conflict and interference. We abuse notation by treating elements of tgt^*, *e.g.*, R and W, as sets of targets.

to by the instance target is included in the class of objects that the any-instance target may refer to. The rule for two any-instance targets is similar.

In Figure 4.20, we extend target overlap to apply to a set of targets. Target overlap needs to account for state aggregation from the mapping of regions of unshared fields. This is accomplished by elaborating the sets of targets *before* comparing them.[9] Specifically, two effects *conflict* if after elaborating their respective targets, one of them writes a target that overlaps with a target read or written by the other. Finally, two expressions *interfere* if they have conflicting effects.

[9]Testing for overlap this way is overly conservative. Using a permissions-based state semantics as discussed in Section 4.9 should enable the use of a less conservative and more intuitive process of testing for overlap.

Chapter 5

Protecting State

A fundamental requirement for assuring the correct use of shared state is the identification of *what state is shared*. This state may encompass multiple fields of an object and even span multiple objects. Chapter 4 describes our use of *regions* to identify what is the state of an object and for aggregating the state of several objects into a single named abstraction of state. Having this groundwork to name state, we can consider protecting portions of state from concurrent access by using associations of locks with state identified by hierarchical regions to answer *how is access to state synchronized*.

Documenting the association between locks and specific portions of state is *not* a new idea. The novelty of our technique is the association of locks with *abstract aggregations of state*. This has important 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. Tool-supported analyses based on documented lock-related design intent effectively address many of the flaws in Java's implementation of the monitor concept; see, for example, [BH99b]. In particular, explicit declaration of shared state and its associated lock enables the assurance of correct access outside of the class that declares the state. Associating locks with hierarchical regions additionally enables program transformations that can systematically alter the granularity of data protection by splitting across subregions (lock-splitting), or vice versa (lock-merging); see Chapter 8.

5.1 Associating Locks with State

Regions are treated like Hoare's *resources* [Hoa71]: as abstract groups of shared state. The programmer identifies a *shared region* and describes how it is to be protected by annotating a class with

`@lock` *lockName* `is` *representation* `protects` *region*

This declares that *region* is potentially shared and thus access to it is mediated via the lock referenced by *representation*. The lock is known by the abstract name *lockName*, hiding the lock's representation from clients. We require *representation* to be either `this` or a `final` field we select to be the canonical reference to our lock object. Immutability via `final` prevents the object used as the lock from changing during the lifetime of the referring object. A special case of a `final` field is the Java class expression *C*`.class`, where *C* is a class name, that mimics a `static` field of type `Class` and evaluates to the `Class` object for class *C*.

In addition to the requirement that the representation of a lock be `final`, there are several other conditions that must be satisfied for the declared locking model to be well formed; these are formally specified in Section 5.11. The representation of the lock that protects a region must be at least as visible as the shared region. To avoid breaking abstraction, this condition may be met by declaring methods that return references to locks; see Section 5.4. *No region may be associated with more than one lock.* Doing so enables race conditions because different accesses to the region could then acquire different locks which does not guarantee mutually exclusive access to the region. This rule has several immediate implications for the locking model: (1) a region cannot be associated with a lock if it has an ancestor region that is associated with a lock; and (2) a `static` region must be protected by a `static` field.

5.1.1 Identifying Locks

We distinguish between lock names and lock identifiers. A *lock name* is declared in a class via the `@lock` annotation, and, for instances, declares a family of locks, one for each member of the class. Thus, the lock name refers to a particular region of the class. Much like a target does for regions, a *lock identifier* identifies a particular lock for a particular instance. A lock associated with a `static` region is identified by the lock name qualified by the name of the class in which it is declared. A lock associated with an instance, *i.e.*, non-`static`, region must be identified with respect to a particular object. An instance lock is thus identified by the pairing of an object-valued expression with a lock name, *e.g.*, `this.fifo.BufLock`, which identifies the lock named `BufLock` of the object referenced by expression `this.fifo`. Instance lock identifiers have the same sort of aliasing problems as instance targets.

5.1.2 The Locking Model

Lock annotations define a model of state protection for the class that is enforced by static analysis. In particular, any field that is a member of the region must be accessed only from within a critical section holding the given lock. 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 representation of the lock must also be accessible. Unprotected accesses to state that is part of this model are revealed as inconsistencies between the stated design intent and source code. It 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. We are currently developing "thread coloring" annotations for thread identification that, among other uses, can address this problem [SGS02].

Fields that are declared to be immutable, `final` in Java, are exempt from the declared locking model. Even if a `final` field is a subregion of a protected region, no lock needs to be held before accessing the field. Fields declared to be `volatile` are also exempt from the declared locking model because volatility expresses the intent that the value of the field can change unpredictably (because it is not in the thread's local memory) and thus it cannot participate in an invariant with another field. Of course, objects referenced through `final` and `volatile` fields are not exempt from the locking model that might be associated with their type.

5.2 Example: Class BoundedFIFO

We return to the class `BoundedFIFO` from Log4j. Previously, in Chapter 4, we annotated the class with regions and effects; the results are shown in Figure 5.1. As a reminder, the class is used to pass `LoggingEvent` objects between two threads. Clients executing in different threads 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

```
/**
 * @methodSet readers = getMaxSize, length, wasEmpty, wasFull, isFull
 * @set readers reads Instance
 */
public class BoundedFIFO {
  /** @unshared
   *  @aggregate Instance into Instance */
  LoggingEvent[] buf;
  int numElts = 0, first = 0, next = 0, size;

  /** Create a new buffer of the given capacity.
   * @writes nothing */
  public BoundedFIFO(int size) {
    if(size < 1) throw new IllegalArgumentException();
    this.size = size;
    buf = new LoggingEvent[size];
  }

  /** Returns <code>null</code> if empty.
   * @writes Instance */
  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.
   * @writes Instance */
  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 5.1: Class `BoundedFIFO` with region and effect annotations.

```
public class PutterClient { ...
  private final BoundedFIFO fifo;
  ...
  public PutterClient(BoundedFIFO bf, ...) { fifo = bf; ... }
  ...
  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 class GetterClient { ...
  private final BoundedFIFO fifo;
  ...
  public GetterClient(BoundedFIFO bf, ...) { fifo = bf; ... }
  ...
  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;
    }
  }
}
```

Figure 5.2: Canonical clients for `BoundedFIFO`.

spanning a series of calls to a `BoundedFIFO`. Figure 5.2 reorganizes `BoundedFIFO` *client* code from Log4j into canonical producer and consumer clients. Our use of lock-related annotations in this chapter make explicit: (1) the delineation of the shared state of an abstract `BoundedFIFO`, (2) the locking conventions for safely accessing the state, and (3) the placement of responsibility to acquire locks.

The region annotations in `BoundedFIFO` make explicit the design intent that each instance of the class has a unique private array that is treated as if it were part of the FIFO instance. We now wish to annotate the additional intent that the state of a FIFO instance is to be protected by the locking on the instance itself. This is done by annotating the class with

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

The implementation of `BoundedFIFO` does not actually acquire this newly declared lock; we address this issue in Section 5.4.2.

5.3 Example: Class `ThreadCache`

We can easily document the locking design intent of in the classes `ThreadCache` and `CachedThread` from the Jigsaw web server. The classes with region and locking annotations are shown in Figure 5.3. What is potentially surprising to the reader of this code is that part of the state of a `CachedThread` object is protected by the thread object itself, but *other parts of the thread's state are protected by a distinguished* `ThreadCache` *object*. The region structure developed in Section 4.7 helps to describe the locking intent. Recall that the class `CachedThread` is parameterized by a region that is used as the parent region of the `next` and `prev` fields. As a result, the "backbone" of the linked list assembled by a `ThreadCache` object, made up of its `freelist` and `freetail` fields, and the `next` and `prev` fields of `CachedThread` objects, resides entirely in cache's `Threads` region.

The lock annotation on line 12 documents the intent that `ThreadCache` objects use themselves as the lock to protect their `Instance` region. In this case, the `Threads` region is a subregion of `Instance` and, as discussed above, spans multiple objects. This annotation, in combination with state aggregation documented by the region annotations, captures the design intent that the `next` and `prev` fields are protected by the `ThreadCache` object that manages the `CachedThread` object of which they are a part. The annotation on line 44 documents the intent that the rest of the state of a thread object is protected by the thread itself. In particular, the region it associates with a lock, `ThreadInfo`, *does not* contain the `next` and `prev` fields.

Explicating the design intent regarding the shared state of `CachedThread` and `ThreadCache` objects makes it less likely that a future maintainer will assume that the `next` and `prev` fields are protected by their respective thread object. Static analysis provides the additional assurance that the implementation does not deviate from the locking model. For example, analysis assures that method `isFree()` correctly accesses state the state of the linked list backbone. Both parameter `t` and field `freetail` are declared to have the annotated type `CachedThread<this.Threads>`. The field references `t.prev` and `freetail.next` are thus, transitively, accesses to the region `Instance` of the method's receiver, and correctly protected because of the `synchronized` modifier on the method; see line 20.

```
1 /**
2  * @region public Threads
3  * @region public CacheInfo
4  */
5 public class ThreadCache {
6   /** @mapInto CacheInfo */
7   protected int threadcount, usedthreads;
8   // [code omitted]
9   /** @mapInto Threads */
10  protected CachedThread /*@<this.Threads>*/ freelist, freetail;
11
12  /** @lock CacheLock is this protects Instance */
13
14  private synchronized
15  CachedThread /*@<this.Threads>*/ createThread() { ...
16    return new CachedThread /*@<this.Threads>*/ (this,...);
17  }
18
19  /** @writes t.ThreadInfo, this.Instance */
20  synchronized boolean
21  isFree(CachedThread /*@<this.Threads>*/ t, ...) {
22    if(!t.isTerminated()) { ... }
23    else { ...
24      t.prev = freetail;
25      if(freetail != null) freetail.next = t;
26      freetail = t;
27      if(freelist == null) freelist = t;
28      usedthreads--; ...
29    } ...
30  } // [rest of class omitted]
31 }
32
33 /** @region public ThreadInfo */
34 class CachedThread /*@<region Backbone>*/ extends Thread {
35   private final ThreadCache cache;
36   /** @mapInto ThreadInfo */
37   private boolean alive;
38   /** @mapInto ThreadInfo */
39   private Runnable runner;
40   // [code omitted]
41   /** @mapInto Backbone */
42   CachedThread /*@<Backbone>*/ next, prev;
43
44   /** @lock ThreadLock is this protects ThreadInfo */
45
46   /** @writes ThreadInfo */
47   synchronized boolean isTerminated() { ... }
48
49   // [Additional synchronized methods follow]
50 }
```

Figure 5.3: Classes `ThreadCache` and `CachedThread` with lock annotations: lines 12 and 44.

5.4 Lock Usage Annotations

A method is usually assumed to be responsible for introducing the critical sections necessary for any shared state that it accesses. This common Java assumption is often relaxed when a `private` method assumes its callers already hold locks. This is useful when the method is likely to be commonly invoked in contexts where the necessary locks are already held. The Javadoc for the `copy` method in class `java.lang.StringBuffer`, for example, states "[This method] should only be called from a `synchronized` method." The Java assertion mechanism includes the ability to test whether a thread holds a particular lock [Blo01b]; we encounter an example of this in Section 7.6. These are informal annotations of design intent: the former is not assurable by a analysis; the latter is not statically assurable, rather it results in a runtime exception when violated.

To enable static assurance of such methods, we use a formal annotation that answers the question *which locks must be held before a method may be invoked*. The same information also answers the converse: *what locks are assumed to be held when this method is invoked*. We refer to this design intent as "locking preconditions," and record it using an annotation on methods:

$\boxed{\texttt{@requiresLock}\ \mathit{lockName}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{lockName}_n}$

This annotation defines an analysis cutpoint. When assuring an implementation of the method, analysis can assume that the named locks are held. The validity of this assumption is preserved by assuring that each invocation of the method occurs in a context where the named locks are in fact held. To preserve modular reasoning, the locking precondition of a method is not allowed to be extended by a reimplementation of the method.

Locking preconditions are in terms of locks of the receiver or static locks. It is not presently allowed to name locks of other objects as preconditions, but an obvious and straightforward extension is to allow the naming of locks of parameters.

In addition to requiring locks, a method can also result in a lock. Because locks are objects in their own right, references to objects used as locks can be method return values. This occurs in the Java Abstract Window Toolkit (AWT) implementation, for example: class `java.awt.Component` has a method `getTreeLock`. This method "Gets this component's locking object (the object that owns the thread synchronization monitor) for AWT component-tree and layout operations." Methods that return locks are useful for hiding the exact representation of a lock from clients. By using a method that is declared to return a specific lock, the field that actually refers to that lock can be kept private to the implementation of the class. Exposing a lock by increasing the visibility of the field that refers to it violates more than just the standard principles of encapsulation as applied to locking concerns. Because it is often the case that the object used as a lock is also used to implement other functionality within the class, requiring the lock reference to be exposed would thus require data abstraction principles to be violated in the service of obtaining assurance about lock-related design intent.

The design intent that a method returns a particular lock of the receiver is recorded using the annotation

`@returnsLock` *lockName*

The return value from such a method can be used in contexts where the lock *lockName* is expected; Section 5.6 discusses this in more detail. A method with a `@requiresLock` annotation must be assured to return the named lock.

5.4.1 Methods `wait` and `notify`

An interesting case of method preconditions are the methods `wait` and `notify`, and their kin, in `java.lang.Object`. These methods are used to implement a condition variable. As such, they require that the lock for the object is held before they are invoked; an exception is thrown at run-time if the lock is not held. We use our `@requiresLock` annotation to assure statically that the appropriate lock is held when they are invoked. In particular, we annotate the class `Object`[1] to introduce a new `public` region `WaitQueue` and associate the lock represented by the receiver with the region:

```
@region public WaitQueue extends All
@lock MUTEX is this protects WaitQueue
```

Methods `wait`, `wait(long)`, `wait(long,int)`, `notify`, and `notifyAll` are all annotated with the lock precondition

```
@requiresLock MUTEX
```

5.4.2 Class `BoundedFIFO` Revisited

For an instance of `BoundedFIFO`, (1) the instance itself is locked to protect its state and (2) clients of the FIFO instance, rather than the FIFO's methods, are expected to acquire that lock. In Section 5.2

[1] The mechanism used for annotating preexisting library code is described in Section 7.3.

we showed how to document the design intent for (1). We now document the design intent for (2). That this is the intent is supported by the observations

- None of the methods in `BoundedFIFO` are `synchronized`
- The clients shown in Figure 5.2 synchronize on the FIFO object before invoking any of its methods.

We record this design intent by annotating each method of `BoundedFIFO` with the locking precondition

```
@requiresLock BufLock
```

Figure 5.4 shows a fully annotated `BoundedFIFO` implementation.

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 locking convention. With the annotation, assurance of the implementation of `BoundedFIFO`'s methods succeeds because each method is analyzed under the assumption that the lock `BufLock` is held. Assurance of the clients succeeds because the block `synchronized(fifo) {...}` can be determined to acquire the lock `fifo.BufLock` necessary to satisfy the preconditions of the methods invoked within the block. Analysis is also able to assure that the precondition of `wait` and `notify`, the lock `fifo.MUTEX`, is satisfied by the client code. The same `synchronized` blocks that satisfy the preconditions for the `BoundedFIFO` methods also satisfy the preconditions for the condition variable because the reference `fifo` also represents the lock `fifo.MUTEX`.

Occasionally it is necessary to reorganize code in the service of annotation. For example, if the implementors of `BoundedFIFO` do not wish to reveal to clients of the class that `BufLock` is `this`, perhaps because they do not want to be committed to using this representation in the future, they could introduce a new method into `BoundedFIFO`:

```
/** @returnsLock 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()) {...}
```

By better encapsulating the representation of `BufLock`, this approach makes it easier to change the representation of `BufLock` in the future. Without this encapsulation, all uses of `BoundedFIFO` instances would have to be found and the lock that clients acquire altered to conform to the new representation. This is clearly a non-local modification, especially if `BoundedFIFO` were part of a library. With this encapsulation however, just a local change to `getLock`—and its associated assurance—is required.

As a caveat to the above, we remark that in the case of locks represented by `this`, the use of lock-returning methods cannot be enforced by visibility constraints on the lock representation because the receiver for a method is always available to the caller. In the case of locks represented by fields of a class, the field can be made private to the implementation and thus the clients will have to use the method to get a reference to the lock because the standard language semantics will prevent the clients from referencing the private field. An appealing general solution to this problem is to require that locks be obtained from method calls only, but this may lead to an awkward coding style, or at least one that is not commensurate with current Java coding practices. We thus opt not to impose this requirement.

5.5 Shared State and Object Construction

When an object is constructed it is responsible for initializing the values of its fields. The `@lock` annotation associates locks with subsets of fields, via regions, that must be held before those fields are read or written, and these ought to apply to the field assignments performed during object construction. Common programming practice in Java, however, is to not acquire locks in the implementation of constructors. In fact, while the Java language allows a method to be declared to be `synchronized`, it is *illegal* to declare a constructor to be `synchronized`. The actions of a constructor are allowed to be fully general, however, and thus there is no general rationale for excusing constructors from complying with accepted thread-safe programming practices. What explains this seeming contradiction between recommendation and practice?

A better question is under what circumstances might an object in the process of being constructed be accessed by multiple threads? This can only happen if a reference to the newly created object, the object referenced by `this` in the implementation of the constructor, is given to another thread during the process of construction. Truly, it is an unusual constructor that allows this to happen. Thus, if the constructor does not leak a reference to the object to another thread, it is impossible for the object to be concurrently accessed during construction. It is thus beneficial to document explicitly the intent that the constructor does not leak the newly constructed object to any other threads. We use the annotation

```
@synchronized
```

on constructors for this purpose. This annotation is a misnomer because the constructor does not acquire locks as a result of the annotation, but is meant to be evocative of `synchronized` methods and to address the common complaint of inexperienced Java programmers that constructors cannot be `synchronized`.

When a constructor so annotated is analyzed, it is assumed that all the programmer-declared locks associated with instance regions of the class are held. Specifically, the lock `MUTEX` defined in `Object` is not assumed to be held because this would incorrectly allow the `wait` and `notify` methods to be invoked outside of a critical section. Locks protecting `static` regions are not assumed to be held because `static` state exists independently of any instances and can thus already have references to it from other threads. It remains for analysis to assure that the newly constructed object is not leaked to any other threads. Our implementation of this analysis is overly conservative. For a `@synchronized` constructor, analysis verifies that the receiver is declared to be `@borrowed`, that is, that *no* aliases to the receiver survive when the constructor returns to its caller. Assurance that the receiver is actually borrowed is performed by the same analysis that assures the correct use of unshared fields [Boy01a], and is beyond the scope of this work. This analysis is overly conservative because it does not consider at all the question of whether the surviving aliases are from other threads. This particular means of assurance is simply a convenient implementation choice, however, and more discerning analyses, such as those of [Bla99, BH99a, CGS^{+}99, WR99] could be used to assure the `@synchronized` annotation.

We observe that short of using a lock external to the object being created, and acquired prior to invocation of the constructor, it is impossible to guarantee that an object will be constructed within a single critical section. This is because attempting to synchronize a constructor by enclosing its body within a `synchronized` block fails to capture the mandatory invocation of a constructor of the superclass. Thus the best that can be achieved by synchronizing the body of a constructor is construction across multiple critical sections, which still allows the possibility of another thread accessing the object in between constructors.

5.5.1 Constructing BoundedFIFO

As an example, consider the constructor of class `BoundedFIFO`. The fields `size` and `buf` are initialized by the constructor, and analysis to assure consistency with the stated locking model will fail to assure the constructor because it does not acquire `this`. The constructor cannot pass the responsibility to its callers via a `@requiresLock` annotation: the caller cannot have a reference to the required lock—*i.e.*, the object being created—because it does not yet exist at the point where it must be acquired. (Constructors can require as a precondition the acquisition of `static` locks.) It is obvious that it is impossible for the `BoundedFIFO` object to be shared while being constructed—the object is clearly thread-local during construction. We thus document the design intent that the constructor does not allow the receiver to escape to another thread, and add the supporting aliasing annotation; the results are in Figure 5.4.

5.6 Identifying Locks

The primary analysis problems for assuring the correct use of shared state are (1) identifying the portion of shared state that is being accessed, and therefore the lock that is required to access it, and (2) identifying the locks that are acquired by `synchronized` statements and methods. The first problem is addressed by the programmer-declared regions of state and programmer-declared lock–region associations. Shared state is accessed via field references and indexing into arrays. The region hierarchy as modified by uniqueness and parameterization aggregations is consulted to determine if an ancestor region of the field is associated with a lock. If such an ancestor region exists, then that region's associated lock is required to access the field. For example, consider the expression `this.buf[this.first]` in the `get` method of `BoundedFIFO`. The field `first` is a subregion of the receiver's `Instance` region, which is associated with the lock named `BufLock` represented by `this`. The array access `this.buf[...]` also accesses the region `Instance` of the receiver because of uniqueness aggregation. Thus the lock required by both accesses is the lock identified by `this.BufLock`. The process of identifying which lock is required for a particular field access is formally described in Section 5.11.

To determine which lock is acquired by a `synchronized` block, an object-valued expression must be converted into the set of locks that it can represent. A single expression can simultaneously represent locks for different regions of the same object because the same representation can be used for many different locks within a class. A field reference e`.f`, where `f` is non-`static`, can additionally represent locks for two different objects: (1) the locks represented by the field `f` of the object referenced by e, and (2) the locks represented by `this` of the object referenced by e`.f`. For a lock expression e (where e may in turn be a field reference expression), the type of e is checked to determine the lock names, if any, whose representation is `this`; for each such lock name l, the lock identifier $e.l$ is added to the set of locks. For a field reference e`.f`, the type of e is consulted to determine the lock names, if any, whose representation is the field `f`. If `f` is `static`, then the trivially constructed lock identifiers are added to the set of possible locks; otherwise, for each such

```
/**
 * @methodSet readers = getMaxSize, length, wasEmpty, wasFull, isFull
 * @methodSet calleeLocked = get, put, readers
 * @lock BufLock is this protects Instance
 * @set readers reads Instance
 * @set calleeLocked requiresLock BufLock */
public class BoundedFIFO {
  /** @unshared
   * @aggregate Instance into Instance */
  LoggingEvent[] buf;
  int numElts = 0, first = 0, next = 0, size;

  /** Create a new buffer of the given capacity.
   * @writes nothing
   * @borrowed this
   * @synchronized */
  public BoundedFIFO(int size) {
    if(size < 1) throw new IllegalArgumentException();
    this.size = size;
    buf = new LoggingEvent[size];
  }

  /** Returns <code>null</code> if empty.
   * @writes Instance */
  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.
   * @writes Instance */
  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 5.4: Class `BoundedFIFO` with region, effect, and locking annotations.

lock name l, the lock identifier e.l is added to the set of locks. A method can return a lock as a result; an expression e.m(...) is similarly converted to lock identifiers based on the lock name in m's @returnsLock annotation.

A static synchronized method is treated as a synchronized block whose lock expression is the pseudo-field class. A synchronized instance method is treated as a synchronized block whose lock expression is this. Lock preconditions declared via @requiresLock annotations must also be converted to lock identifiers; this is a straightforward process.

The *lock context* for an expression is the set of locks that are held when the expression is evaluated. It is the union of the locks acquired by all surrounding synchronized blocks (also accounting for the possibility that the expression is in the body of a synchronized method) and of the locks assumed to be acquired by any @requiresLock precondition on the method in which the expression appears.

The general process for checking that a field is accessed according to the locking model is (1) compute the lock identifier for the field reference; (2) compute the lock context for the expression; and (3) determine if the lock identifier from (1) *must* name any of the locks in the lock context. In general, alias analysis is required to compare two instance locks. We sidestep this problem in our implementation by restricting the syntactic form of lock expressions. We require them to be *final expressions*, by which we generally mean an expression whose value is constant. Such an expression can be used as a name for an object throughout the scope in which it is defined with out having to worry that different uses of the expression refer to different objects. The use of a final local variable, including this, is a final expression; a field reference e.f is a final expression if e is a final expression and f is a final field; a method invocation e.m(...) is a final expression if e is final expression and m is a method with a @returnsLock annotation. A program can be converted to use only final expressions by introducing final local variables.

Only lock expressions that are final expressions are converted into locks. Our prototype analysis tool outputs warnings when it encounters non-final lock expressions. Our implementation compares locks by first comparing the lock names, and then checking if the object expressions are syntactically identical. We do not require that shared state be accessed only via final expressions. However, state that is not accessed via a final expression will never be assured to have the required lock held. The details of final expressions and lock comparisons are in Section 5.11.

5.7 Protecting *References*

In Java, a class-typed field actually has as its value a *reference* to an object. As a result, it is in general the case that an object referenced by the field of one object can also be referenced by a field of another distinct object. In other words, unlike C++, Java does not provide an explicit mechanism for incorporating the structure of one object into the structure of another. This situation motivates the use of uniqueness and `@unshared` fields within our region and effects system in Chapter 4. This problem of potential aliases also impacts how state is protected. A lock associated with a region guards the *values* of the fields in that region. In case of class-typed fields, therefore, the reference only is protected, not the object that is referenced. This is not a defect in the locking meta-model: it is in general unsafe to use the lock that protects the field to also protect the referenced object because to do so in the absence of additional design information could lead to multiple locks protecting the same object.

Sometimes this is what is desired by the programmer because it is assumed the referenced object is already thread-safe. That is, its implementation performs the actions necessary to protect the invariants of class instances—*e.g.*, by being immutable or using locks—or otherwise describes to clients what they must do to preserves an instance's invariants. These are issues of concurrency policy and are further discussed in Chapter 6. The problem with this situation is when it is *not* what the programmer intends. It is a problem because simply verifying that the reference to an object is correctly accessed gives no assurance about the thread-safety or lack thereof of the referenced object. The danger is that because our approach is incremental and does not require all classes to be thread-safe at all times, unlike for example, RACEFREEJAVA [FF00], Guava [BST00], and Parameterized RACEFREEJAVA [BR01], assuring the correct use of a field can be misleading. Instances of the assured class could still participate in race conditions over the object that they reference because the referenced object is shared and not thread-safe.

The issue for us, then, is how to make analysis less misleading under these circumstances. Our solution is to analyze the use of reference-valued fields that are associated with a lock and issue warnings when the usage appears to make use of a potentially shared object that may be non–thread-safe. These warnings are not assurance failures: they do not reflect inconsistencies between source code and annotated design intent. Rather they are requests from the tool for more information about design intent; they highlight areas where the additional assurance enabled by additional design intent may be beneficial. The object referenced by the protected field can be dereferenced to access a field, or to invoke a method. These uses are analyzed differently. To be more concrete, we consider a field reference *e*.`f` where field `f` with declared type `C` is within a shared region of the object of type `D` evaluated to by expression *e*.

For a field reference *e*`.f.g` we try to verify that field `g` is known to be protected. If `g` is `final` or `volatile` then it does not require protection, and analysis does not issue a warning. Otherwise analysis checks if within class `C` the field `g` is part of a shared region. If so, then no warning is issued because there is a lock associated with this field, and the standard lock analyses will assure the correct use of the field. Alternatively, field `f` may be `@unshared` and the field `g` aggregated into a protected region of the `D` instance referenced by *e*. Again, the analysis for assuring the correct use of fields will handle this case correctly. It is only if `g` is not declared to be protected that a warning is issued.

For a method invocation *e*`.f.m(...)` we first check if the field `f` is `@unshared`. If it is, then the invocation does not need to be additionally protected because the referenced object cannot be accessed by multiple threads without going through field `f`, which is already protected. If the field is not declared to be `@unshared`, then we try to determine if objects of class `C` are thread-safe. Really what we want to know is whether the method `m` can be invoked without any client-side locking. But the absence of a `@requiresLock` annotation on `m`'s declaration *does not* indicate design intent one way or the other—the concurrency policy of the class needs to be consulted. It is enough to know that instances of the type "take care of themselves." The annotation

```
@selfProtected
```

on class and interface declarations declares this intent. This annotation and concurrency policy issues in general are discussed in Chapter 6. If `f` is not `@unshared` and type `C` is not annotated with `@selfProtected` then a warning is issued.

We emphasize (1) that these warnings do not correspond to inconsistencies between code and design intent; (2) that these warnings are intending to provoke the programmer into *providing more design intent*; and (3) that the analyses that support these warnings are not intended to be comprehensive, but are heuristics intended to prevent the programmer from being misled by positive assurance results under certain circumstances.

5.7.1 `BoundedFIFO`'s `buf` Array

We use `BoundedFIFO` as an example for why these warnings are interesting. In this example, we assume that the field `buf` has not been previously declared to be `@unshared`. Indexing into the array, as done in methods `get` and `put` would then *not* be covered by the locking model declared by the annotation `@lock BufLock is this protects Instance`. As the code is currently written, the consequences of this lack of explicit protection are obscured because the array cannot be indexed without first dereferencing the field `buf`, which *is* explicitly protected. Suppose, however, that the field `buf` is declared to be `final` and thus not protected by the lock associated with `Instance`.

When analyzing the code in this modified state, with the uniqueness aggregation removed and with `buf` declared to be `final`, our declared locking model

- Specifically excludes `buf` from the protection of the lock associated with region `Instance`.
- Says nothing about how the contents of the array referenced by `buf` are to be protected. In particular, the silence of the model on this point cannot be interpreted to mean that the contents of the array are possibly shared and protected, nor can it be interpreted to mean that the contents of the array are not intended to be shared with other threads and thus do not require protection.

Analysis thus has nothing to assure regarding field `buf` or the array it references. But, as pointed out above, this is misleading because the array object could be aliased and should be protected. We observe that there are few scenarios where it would be useful to alias the array referenced by `buf`. *This is the point*: aliasing the array would violate the design intent for the class and this intent should be made explicit. Analysis augmented with the above heuristics identifies uses of the array as suspicious because the class `BoundedFIFO` contains some concurrency intent, the protection of the `Instance` region. One way to address these warnings is to introduce protection into the class of the referenced object, because arrays are a primitive object type, this is not possible. Instead the array must be aggregated into a protected region of the FIFO object, as originally done in the example.

5.8 Protecting Aggregated Objects

We show in Section 5.2 how to use uniqueness aggregation to protect the contents of the `buf` array in the `BoundedFIFO` example. Aggregation in this case maps the elements of the array, the region `[]`, into the `Instance` region of the `BoundedFIFO` object; the mapping is indirect, via the `Instance` region of the array object. As a result of this mapping, within the implementation of `BoundedFIFO`, effects on target `this.buf.[]` are treated as effects on target `this.Instance` and thus must occur within a critical section synchronized on `BufLock`. Thus, in general, to check that an access to a region that is aggregated into the region of another object conforms to the locking model, locking analysis merely has to consult the aggregation mappings, and apply the locking model as applied to the incorporating region. While this description provides the intuition behind the interaction of lock analysis and uniqueness aggregation, there are many complicating issues that must be addressed.

Aggregating the contents of an array represents the simplest and most straightforward case to handle because (1) arrays have no methods, and as a result (2) all changes to the state of an array occur by direct access to its state, and (3) arrays do not attempt to protect themselves. We now consider the issue more generally. Suppose that class `Owner` has an unshared field `u` that refers to an object of class `C`, and that region `Src` of class `C` is mapped into region `Dest` of class `Owner`. The superregion of `u` is irrelevant to this example. If `Dest` is not associated with a lock, then `Src` is protected, or not, according to the locking model of class `C`, and aggregation has no role to play regarding the protection of shared state. On the other hand, if `Dest` is associated with a lock, then *within the implementation of* `Owner`, the intuitive semantics of state aggregation imply that accesses to the region identified by target `this.u.Src` must occur only when the lock associated with region `Dest` is held. Furthermore, any lock associated with region `Src` can be ignored—this is safe because the uniqueness of field `u` guarantees that the only way to access the object is through its associated synchronization-providing `Owner` object

Applying `Owner`'s locking model to aggregated `C` objects, however, is tricky because `C` is implemented without regard to how it may be aggregated into other objects. This is fundamentally an issue of encapsulation, scalability, and cutpoints. Returning to our intuitive explanation from above, to check the implementation of `Owner`, analysis must rely on the *effects* of operations on the `C` object referenced by field `u`. If the operation may affect the region identified by target `this.u.Src`, then the operation is considered to affect the region identified by `this.Dest`, and the appropriate lock must be held. Direct accesses to fields in the aggregated region, *e.g.*, `this.u.f`, where `f` is a subregion of `Src`, are simply direct producers of effects. For method calls, *e.g.*, `this.u.m()`, analysis must rely on the method's declared effects; determining whether `this.u.Src` is affected by the method is a problem solved in Chapter 4. It may be the case that the declared effects are overly broad and force the lock for `Dest` to be acquired unnecessarily: this is an unavoidable consequence of encapsulation.

5.9 Escaping Protection

It is possible to escape the locking model by abusing upcasts. This is a consequence of our interest in incrementality: we do not require that all fields of a class be protected by a lock, unlike, for example RACEFREEJAVA [FF00] and Guava [BST00]. Thus a class can associate an inherited region with a lock as long as that region is not already associated with a lock. The intent here is clear: uses of the region in instances of the ancestor class do not require protection, but uses of the region in instances of the subclass do. An instance of the protection-requiring subclass may be accessed through a reference whose type is that of the unprotected superclass, which causes the protection information to be lost. This is a problem only if the reference is used to directly access fields of the object that belong to the now-forgotten shared region. Method calls via the upcast reference are *not* affected because the preconditions of a method cannot be strengthened by subclasses.

A possible solution to this problem is to propagate region–lock associations up the class hierarchy. This is unreasonable because multiple subclasses of a class could declare different locking models, but more importantly because "backwards" propagation of information along the class hierarchy violates principles of modularity and abstraction and is counterintuitive. A more reasonable approach is for analysis to issue a warning when a reference to an object of a class that adds protection to an inherited region is explicitly or implicitly upcast. Our prototype tool does not check for such situations.

5.10 Related work

Warlock [Ste93] is a static analysis tool that checks for the inconsistent use of locks in C programs written for Solaris. The programmer does not annotate the intended lock–state associations. Instead the tool determines which locks are held when a variable is accessed. Those variables that are not consistently accessed are subject to data races. Annotations are used, however, to mark segments of code where locking is not needed, to declare lock preconditions on functions, and to identify functions that acquire or release locks as side effects.

Both ESC/Java [DLNS98, LNS00] and RACEFREEJAVA [FF00] 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 [FA99a, FA99b]. While [FA99b] 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. As discussed in Section 4.10, ESC/Java does not support state aggregations. Classes in RACEFREEJAVA may be parameterized by locks, which is similar to our state aggregation, but the system has no formal model of named object state. The lock that protects access to the elements of an array may be identified [FQ03a], but how the identity of the array is preserved is not described. RACEFREEJAVA supports thread-local classes which are not currently expressible in our system. RACEFREEJAVA has been extended [FF01] 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. They do not assure anything as a result of this annotation; it is merely a flag for turning off "false alarms." Neither ESC/Java nor RACEFREEJAVA can represent unique pointers.

As described in Section 4.10, object ownership [CPN98] models transitively aggregate the state of an object into the state of its owner. Guava [BST00] and Parameterized RACEFREEJAVA [BR01] use object ownership as the basis for protecting shared state. 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 4.5 in their current form, cannot be expressed in object-ownership–based systems. Guava is a dialect of Java without a general synchronization construct. Instead, classes belong to one of three categories: sharable monitors, sharable deep-copied values, and unsharable objects. Object ownership prevents sharing of objects: each object has exactly one owning monitor or value, and cannot change its owner. Guava sacrifices flexibility for safety, whereas we ultimately desire to enable implementation flexibility and evolution while maintaining safety, and to do this for existing code, as in our case studies.

Parameterized RACEFREEJAVA uses a type-system to enforce locking conventions. It is distinguished by allowing 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 owners.

The Vault programming language [DF01] 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

$$\begin{array}{rcl}
\mathit{defn} & ::= & \textsf{class } \mathit{cn} \textsf{ extends } c\ \{\mathit{region}^*\ \mathit{lockdef}^*\ \mathit{field}^*\ \mathit{meth}^*\} \\
\mathit{region} & ::= & \textsf{region } \mathit{argn} \textsf{ in } \mathit{argn} \\
\mathit{lockdef} & ::= & \textsf{lock } \mathit{ln} \textsf{ is } \mathit{lock} \textsf{ protects } \mathit{rgn} \\
\mathit{lock} & ::= & \textsf{this} \mid \mathit{fd} \\
\mathit{field} & ::= & \mathit{shared_field} \mid \mathit{unshared_field} \\
\mathit{shared_field} & ::= & \mathit{mod_t}\ \mathit{fd} \textsf{ in } \mathit{argn} = e \\
\mathit{unshared_field} & ::= & \textsf{unshared } \mathit{mod_t}\ \mathit{fd} \textsf{ in } \mathit{argn} \textsf{ aggregate } \mathit{agg}^+ = e \\
\mathit{agg} & ::= & \mathit{rgn} \textsf{ into } \mathit{argn} \\
\mathit{meth} & ::= & t\ \mathit{mn}_{\mathit{lbl}}(\mathit{arg}^*) \textsf{ requires } \mathit{ln}^*\ [\textsf{returns } \mathit{ln}]_{\mathrm{opt}}\ \{\mathit{body}\}
\end{array}$$

The disjoint name spaces are extended with the set of abstract region names and lock names. The set of region names is the set of abstract region names together with the set of field names.

$\mathit{ln} \in$ lock names $\quad \mathit{rgn}_{\mathrm{a}} \in$ abstract region names $\quad \mathit{rgn} \in$ field names $\cup$ abstract region names

Figure 5.5: Syntax modifications to FLUIDJAVAwith labeled expressions for regions and locks.

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, or enforcing object protocols, it cannot be assumed that any particular key defines a field–lock relationship.

5.11 Safe Locking in FLUIDJAVA

We present named locks as an extension of FLUIDJAVA with labels, and not as a extension of the language with effects. This presentation does not consider uniqueness aggregation when determining which lock is required to access a field, and thus we do not require effects. Removing effects from consideration simplifies the presentation of locking concerns.

The syntax of FLUIDJAVA is modified as shown in Figure 5.5. Lock declarations associating a named lock with a representation, either the receiver or a field of the class, and a region of state are added to class declarations. Methods must now list zero or more required locks that must be held by the caller in order to invoke the method. A method may optionally have a returns clause declaring that the object returned by the method is the named lock for the receiver.

We carry over the following notational machinery from FLUIDJAVA extended with effects (see Figure 4.7):

- The predicates $\mathbf{RegionsOnce}(P)$, $\mathbf{WFRegions}(P)$, and $\mathbf{CompleteRegions}(P)$ for regions, and $\mathbf{AggregationsOK}(P)$ and $\mathbf{MapOK}$ for aggregation mappings.

$\mathbf{LocksOnce}(P) \equiv \forall ln, ln' : \textsf{class} \cdots \{\cdots \textsf{lock}\ ln \cdots \textsf{lock}\ ln' \cdots\}\ \text{is in}\ P \Rightarrow ln \neq ln'$
$c.ln \Subset^{c} c \Leftrightarrow \textsf{class}\ c \cdots \{\cdots \textsf{lock}\ ln \cdots\}\ \text{is in}\ P$
$c'.ln \in^{c} c \Leftrightarrow c'.ln \Subset^{c} c' \wedge c \leq^{c} c'$
$ln \overset{c}{\mapsto} \hat{c}.ln \Leftrightarrow \exists \hat{c} : \hat{c}.ln \in^{c} c$
$c.ln\ \textsf{is this} \Leftrightarrow \textsf{class}\ c \cdots \{\cdots \textsf{lock}\ ln\ \textsf{is this} \cdots\}\ \text{is in}\ P$ $c.ln\ \textsf{is}\ c'.fd \Leftrightarrow (\textsf{class}\ c \cdots \{\cdots \textsf{lock}\ ln\ \textsf{is}\ fd \cdots\}\ \text{is in}\ P) \wedge (\exists c',\ \hat{c} : \langle c'.fd, \textsf{final}\ \hat{c}\rangle \in^{c} c)$
$c.ln\ \textsf{protects}\ c'.rgn \Leftrightarrow (\textsf{class}\ c \cdots \{\cdots \textsf{lock}\ ln\ \textsf{is}\ lock\ \textsf{protects}\ rgn \cdots\}\ \text{is in}\ P) \wedge rgn \overset{c}{\mapsto} c'.rgn$
$\mathbf{WFLockDefs}(P) \equiv$ $\left(\begin{array}{l} \forall c, ln, fd, rgn : \textsf{class}\ c \cdots \{\cdots \textsf{lock}\ ln\ \textsf{is}\ fd\ \textsf{protects}\ rgn \cdots\}\ \text{is in}\ P \\ \Rightarrow \exists c', c'' : c.ln\ \textsf{is}\ c'.fd\ \wedge\ c.ln\ \textsf{protects}\ c''.rgn \end{array}\right)$ $\wedge\ (\forall c, ln, rgn : \textsf{class}\ c \cdots \{\cdots \textsf{lock}\ ln\ \textsf{is this protects}\ rgn \cdots\}\ \text{is in}\ P \Rightarrow \exists c'' : c.ln\ \textsf{protects}\ c''.rgn)$
$\mathbf{NoShadowing}(P) \equiv \ldots \wedge\ (c.ln \Subset^{c} c \Rightarrow \nexists c' : (c \neq c' \wedge c \leq^{c} c' \wedge c'.ln \Subset^{c} c'))$
$\mathbf{ProtectedOnce}(P) \equiv$ $\forall c, c', rgn, ln : (c.ln\ \textsf{protects}\ c'.rgn) \Rightarrow (\nexists \hat{c}, \hat{c}', rgn', ln' : (c'.rgn \leq^{r} \hat{c}'.rgn' \wedge \hat{c}.ln'\ \textsf{protects}\ \hat{c}'.rgn'))$
$\mathbf{LocksOK}(P) \equiv \forall c, mn, ln : \textsf{class}\ c \cdots \{\cdots mn(\cdots) \cdots ln \cdots \{\cdots\} \cdots\}\ \text{is in}\ P \Rightarrow ln \overset{c}{\mapsto} c'.ln$
$\langle mn, t_1 \ldots t_n \rightarrow t, \{ln_1 \ldots ln_m\}, \emptyset, (x_1 \ldots x_n), e\rangle \Subset^{c} c$ $\Leftrightarrow \textsf{class}\ c \cdots \{\cdots t\ mn_{lbl}([\textsf{final}]_{\text{opt}}\ t_1\ [x_1]_{lbl_1} \ldots [\textsf{final}]_{\text{opt}}\ t_n\ [x_n]_{lbl_n})$ $\textsf{requires}\ ln_1 \ldots ln_m\ \{e\} \cdots\}\ \text{is in}\ P$ $\langle mn, t_1 \ldots t_n \rightarrow t, \{ln_1 \ldots ln_m\}, \{ln'\}(x_1 \ldots x_n), e\rangle \Subset^{c} c$ $\Leftrightarrow \textsf{class}\ c \cdots \{\cdots t\ mn_{lbl}([\textsf{final}]_{\text{opt}}\ t_1\ [x_1]_{lbl_1} \ldots [\textsf{final}]_{\text{opt}}\ t_n\ [x_n]_{lbl_n})$ $\textsf{requires}\ ln_1 \ldots ln_m\ \textsf{returns}\ ln'\ \{e\} \cdots\}\ \text{is in}\ P$
$\mathbf{ClassMethodsOK}(P) \equiv \forall c, c', e, e', mn, t_i, t_i', t, t', L, L', L_{ret}, L'_{ret}, V, V' :$ $(\langle mn, t_1 \ldots t_n \rightarrow t, L, L_{ret}, V, e\rangle \Subset^{c} c \wedge \langle mn, t'_1 \ldots t'_n \rightarrow t', L', L'_{ret}, V', e'\rangle \Subset^{c} c')$ $\Rightarrow ((t = t' \wedge t_i = t_i' \wedge L = L' \wedge L_{ret} = L'_{ret} \wedge V = V') \vee (c \not\leq^{c} c'))$

Figure 5.6: Additional predicates and relations for checking FLUIDJAVA extended with locks. Metavariable L is a set of lock names.

- The relations $\prec^{r}$, $\leq^{r}$, and $\mathcal{M}^{c}_{fd}$.
- The triple $rgn \overset{c}{\mapsto} c'.rgn$.
- The region-related extensions to $\Subset^{c}$ and $\in^{c}$.

Additional predicates and relations related to locks are defined in Figure 5.6. $\mathbf{LocksOnce}(P)$ insures a lock name is unique within a class declaration. The relations $\Subset^{c}$ and $\in^{c}$ are extended to include lock names. The triple $ln \overset{c}{\mapsto} \hat{c}.ln$ converts a lock name into a qualified lock name relative to the class c. The new relation is maps a lock to its representation; when the representation is a field, the relation is only defined if the field is final and a reference to an object, *i.e.*, something that can be locked. Similarly, the new relation protects maps a lock name to the region it protects, and is only defined if the region exists in the class in which the lock is declared. The predicate $\mathbf{WFLockDefs}(P)$ is true only when all the lock declarations in a class are well formed, that is, have defined is and protects mappings. Predicate $\mathbf{NoShadowing}(P)$ is extended to prevent lock names from being shadowed. Predicate $\mathbf{ProtectedOnce}(P)$ makes sure that if a region is protected, it is not a subregion of another protected region.

Predicate $\mathbf{LocksOK}(P)$ ensures that all the locks named in requires and returns clauses are present in the class in which the method is declared. The method structure in relation $\Subset^{c}$ is altered to include the set of required locks and the return lock, if any; relation $\in^{c}$, not shown, is updated in the obvious manner. A method can only return a single lock, but we use a set of cardinality zero or one to more easily handle the case when the method is not declared to return a lock. We update $\mathbf{ClassMethodsOK}(P)$ to require that method overriding preserve lock preconditions and returns clauses.

5.11.1 Locks and Aliasing

Locks are represented as a pair of an expression with a lock name, *e.g.*, $\langle \mathsf{this}.\mathrm{f}, \mathrm{c.lock}\rangle$; this lock is the lock named "lock" declared in class c of the object referenced by expression this.f. In this way, locks are similar to instance targets, and have the same aliasing problems. We resort to syntactic restrictions similar to those of [FF00] to avoid having to resolve lock aliases. An expression that is interpreted as a lock must be a *final expression*, by which we generally mean an expression whose value is constant. Such an expression can be used as a name for an object throughout the scope in which it is defined with out having to worry that different uses of the expression refer to different objects. We require that the lock expression in a synchronized block be a final expression. Final expressions are formally defined in the next section, but in general are built from final local variables and accesses to final fields.

5.11.2 Type Rules for Locks

We must modify the typing judgment for expressions to incorporate the current *lock context* or set of locks known to be held when the expression is evaluated. We also introduce a new judgment for typing *final expressions* whose results also include the set of locks that the expression *must* represent:

$$P; E; C \vdash e : t$$
$$P; E; C \vdash_{\text{final}} e : t \ \mathbf{as}\ L$$

Lock context C and lock result L are sets of locks $\{\langle e', c.ln\rangle\}$.

The well-formedness rules rules for FLUIDJAVA extended with locks are shown in Figure 5.7. Programs are checked against the additional lock-related predicates and the program body is evaluated with an initially empty lock context. For each method, its body is checked using the locks declared to be required as the initial lock context. The lock names are converted to locks of the

PROG

$$\frac{\begin{array}{c}\textbf{ClassOnce}(P) \quad \textbf{RegionsOnce}(P) \quad \textbf{MethodsOnce}(P) \quad \textbf{NoShadowing}(P)\\ \textbf{CompleteClasses}(P) \quad \textbf{WFClasses}(P) \quad \textbf{ClassMethodsOK}(P) \quad \textbf{UniqueLabels}(P)\\ \textbf{WFRegions}(P) \quad \textbf{CompleteRegions}(P) \quad \textbf{AggregationsOK}(P) \quad \textbf{LocksOnce}(P) \quad \textbf{WFLockDefs}(P)\\ \textbf{ProtectedOnce}(P) \quad \textbf{LocksOK}(P) \quad P = \mathit{defn}_1 \ldots \mathit{defn}_n\ e \quad P \vdash \mathit{defn}_i \quad P;\emptyset;\emptyset \vdash e : t\end{array}}{\vdash P : t}$$

CLASS

$$\frac{P, cn \vdash \mathit{field}_i \qquad P, cn \vdash \mathit{meth}_i}{P \vdash \textsf{class}\ cn\ \textsf{extends}\ c' \cdots \{\cdots \mathit{field}_1 \ldots \mathit{field}_j\ \mathit{meth}_1 \ldots \mathit{meth}_k\}}$$

FIELD

$$\frac{P \vdash t \qquad P;\emptyset;\emptyset \vdash e : t}{P, c \vdash [\textsf{final}]_{\text{opt}}\ t\ \mathit{fd}\ \textsf{in}\ \mathit{rgn} = e}$$

UNSHAREDFIELD

$$\frac{P \vdash c' \qquad P;\emptyset;\emptyset \vdash e : c' \qquad \textbf{MapOK}(P, \mathcal{M}^{c}_{\mathit{fd}}, c')}{P, c \vdash \textsf{unshared}\ [\textsf{final}]_{\text{opt}}\ c'\ \mathit{fd}\ \textsf{in}\ \mathit{rgn}\ \textsf{aggregate}\ \mathit{agg}_1 \ldots \mathit{agg}_n\ = e}$$

METHOD

$$\frac{\begin{array}{c}P \vdash t \qquad P \vdash \mathit{mod\_t}_i \qquad E = \textsf{final}\ c\ \textsf{this}, \mathit{mod\_t}_1\ x_1, \ldots, \mathit{mod\_t}_n\ x_n\\ P; E; \{\langle \textsf{this}, c_i.\mathit{ln}_i\rangle \mid \mathit{ln}_i \overset{c}{\mapsto} c_i.\mathit{ln}_i\} \vdash e : t \qquad \mathcal{E}_P \supseteq \{(\mathit{lbl}, E), (\mathit{lbl}_1, E), \ldots, (\mathit{lbl}_n, E)\}\end{array}}{P, c \vdash t\ \mathit{mn}_{\mathit{lbl}}(\mathit{mod\_t}_1\ [x_1]_{\mathit{lbl}_1} \ldots \mathit{mod\_t}_n\ [x_n]_{\mathit{lbl}_n})\ \textsf{requires}\ \mathit{ln}_1 \ldots \mathit{ln}_m\ \{e\}}$$

METHODRETURNSLOCK

$$\frac{\begin{array}{c}P \vdash t\\ P \vdash \mathit{mod\_t}_i \qquad E = \textsf{final}\ c\ \textsf{this}, \mathit{mod\_t}_1\ x_1, \ldots, \mathit{mod\_t}_n\ x_n \qquad P; E; \{\langle \textsf{this}, c_i.\mathit{ln}_i\rangle \mid \mathit{ln}_i \overset{c}{\mapsto} c_i.\mathit{ln}_i\} \vdash_{\text{final}} e : t\ \textbf{as}\ L\\ \mathit{ln} \overset{c}{\mapsto} \hat{c}.\mathit{ln} \qquad \{\langle \textsf{this}, \hat{c}.\mathit{ln}\rangle\} \subseteq L \qquad \mathcal{E}_P \supseteq \{(\mathit{lbl}, E), (\mathit{lbl}_1, E), \ldots, (\mathit{lbl}_n, E)\}\end{array}}{P, c \vdash t\ \mathit{mn}_{\mathit{lbl}}(\mathit{mod\_t}_1\ [x_1]_{\mathit{lbl}_1} \ldots \mathit{mod\_t}_n\ [x_n]_{\mathit{lbl}_n})\ \textsf{requires}\ \mathit{ln}_1 \ldots \mathit{ln}_m\ \textsf{returns}\ \mathit{ln}\ \{e\}}$$

Figure 5.7: Well formedness rules for FLUIDJAVA extended with locks.

receiver using the triple $\mathit{ln} \overset{c}{\mapsto} c'.\mathit{ln}$. In this way, the body is evaluated assuming that the method will be called correctly. If a method has a returns clause, the lock declared to be returned is checked against the set of locks that the body of the method is known to represent. The body of the method must be a final expression in this case. In real Java, we would instead check the value of each `return` statement in the method's body, and require that each return statement return a final expression.

Most of the rules for typing expressions just propagate the lock context without using it or altering its contents. The interesting cases are those expressions that can be interpreted as a locks or that require lock information. Figure 5.8 shows the rules for reading a local variable. The first rule VAR is the standard rule for variable use. But the second rule is the first of several judgments defining final expressions. A final local variable is a final expression. The locks represented by the expression are derived from those lock names declared in the type of the variable that are represented by the object itself, *i.e.*, this. This lock generation process is embodied in the helper function $\textbf{objAsLocks}(P, e, t)$, which returns the empty set when t is a non-object type.

More interesting are the rules for field access, shown in Figure 5.9, because such expressions

$$\mathbf{objAsLocks}(P, e, t) \equiv \begin{cases} \emptyset & \text{when } t \in \{\mathsf{int}, \mathsf{boolean}\} \\ \{\langle e, c.\mathit{ln}\rangle \mid t \leq^{\mathrm{c}} c \wedge c.\mathit{ln} \text{ is } \mathsf{this}\} & \text{otherwise} \end{cases}$$

$$\textsc{Var}\ \frac{E = E_1, [\mathsf{final}]_{\mathrm{opt}}\ t\ x, E_2}{P; E; C \vdash x : t} \qquad \textsc{VarFinalExpr}\ \frac{E = E_1, \mathsf{final}\ t\ x, E_2}{P; E; C \vdash_{\mathrm{final}} x : t \textbf{ as } \mathbf{objAsLocks}(P, x, t)}$$

Figure 5.8: Rules for reading local variables.

$$\mathbf{lockFor}(P, e, c.\mathit{fd}, t) \equiv \{\langle e, \hat{c}.\mathit{ln}\rangle \mid t \leq^{\mathrm{c}} \hat{c} \wedge c.\mathit{fd} \leq^{\mathrm{r}} c'.\mathit{rgn} \wedge \hat{c}.\mathit{ln} \text{ protects } c'.\mathit{rgn}\}$$
$$\mathbf{fdAsLocks}(P, e, c.\mathit{fd}, t) \equiv \{\langle e, c'.\mathit{ln}\rangle \mid t \leq^{\mathrm{c}} c' \wedge c'.\mathit{ln} \text{ is } c.\mathit{fd}\}$$

$$\textsc{Get}\ \frac{P; E; C \vdash e : c \qquad \langle c'.\mathit{fd}, t\rangle \in^{\mathrm{c}} c \qquad \mathbf{lockFor}(P, e, c'.\mathit{fd}, c) \subseteq C}{P; E; C \vdash e.\mathit{fd} : t} \qquad \textsc{GetFinal}\ \frac{P; E; C \vdash e : c \qquad \langle c'.\mathit{fd}, \mathsf{final}\ t\rangle \in^{\mathrm{c}} c}{P; E; C \vdash e.\mathit{fd} : t}$$

$$\textsc{GetFinalExpr}\ \frac{P; E; C \vdash_{\mathrm{final}} e : c \textbf{ as } L \qquad \langle c'.\mathit{fd}, \mathsf{final}\ t\rangle \in^{\mathrm{c}} c}{P; E; C \vdash_{\mathrm{final}} e.\mathit{fd} : t \textbf{ as } \mathbf{fdAsLocks}(P, e, c'.\mathit{fd}, c) \cup \mathbf{objAsLocks}(P, e.\mathit{fd}, t)}$$

$$\textsc{Set}\ \frac{P; E; C \vdash e : c \qquad \langle c'.\mathit{fd}, t\rangle \in^{\mathrm{c}} c \qquad P; E; C \vdash e' : t \qquad \mathbf{lockFor}(P, e, c'.\mathit{fd}, c) \subseteq C}{P; E; C \vdash e.\mathit{fd} = e' : t}$$

Figure 5.9: Rules for expressions that access fields.

can both be locks and require the holding of a lock. The function $\mathbf{lockFor}(P, e, c.\mathit{fd}, t)$ gives the lock required, if any, to access the field $c.\mathit{fd}$ of the object referenced by e of type t in program P by finding an ancestor region of $c.\mathit{fd}$ that is associated with a lock defined in t. Although the result is a set, the set must have a cardinality of either zero, if the field is not associated with a lock, or one, if the field is associated with a lock. A set is used to make dealing with the absence of a lock easier. Reading from a non-final field may require lock. The current lock context is checked to see if it contains the required lock. Reading from a final field is similar except that no lock is required to access it. The required lock is checked when writing to a field.

Reading from a final field is a final expression if and only if the subexpression evaluating to the dereferenced object is a final expression. The field reference can represent locks from two objects: (1) locks represented by the field fd of the object e, handled by the helper function $\mathbf{fdAsLocks}(P, e, c.\mathit{fd}, t)$; and (2) locks represented by this of the object $e.\mathit{fd}$, handled by $\mathbf{objAsLocks}(P, e.\mathit{fd}, t)$.

The rules for method invocation are shown in Figure 5.10. A method can only be invoked if the locks it requires are present in the current lock context. In the case of normal method invocation the

$$\textbf{locksFromSet}(P, L, e, c) \quad \equiv \quad \{\langle e, c'.ln\rangle \mid ln \in L \wedge ln \overset{c}{\mapsto} c'.ln\}$$

INVOKE

$$\frac{P; E; C \vdash e : c \qquad P; E; C \vdash e_i : t_i \qquad \langle mn, t_1 \ldots t_n \to t, L_{req}, L_{ret}, V, e_b\rangle \in^{c} c \qquad \textbf{locksFromSet}(P, L_{req}, e, c) \subseteq C}{P; E; C \vdash e.mn(e_1 \ldots e_n) : t}$$

INVOKEFINALEXPR

$$\frac{P; E; C \vdash_{\text{final}} e : c \textbf{ as } L \qquad P; E; C \vdash e_i : t_i \qquad \langle mn, t_1 \ldots t_n \to t, L_{req}, L_{ret}, V, e_b\rangle \in^{c} c \qquad \textbf{locksFromSet}(P, L_{req}, e, c) \subseteq C}{P; E; C \vdash_{\text{final}} e.mn(e_1 \ldots e_n) : t \textbf{ as locksFromSet}(P, L_{ret}, e, c)}$$

SUPER

$$\frac{P; E; C \vdash_{\text{final}} \mathsf{super}.mn(e_1 \ldots e_n) : t \textbf{ as } L}{P; E; C \vdash \mathsf{super}.mn(e_1 \ldots e_n) : t}$$

SUPERFINALEXPR

$$\frac{\begin{array}{c} P; E; C \vdash \mathsf{this} : c' \qquad c' \prec^{c} c \qquad \langle mn, t_1 \ldots t_n \to t, L_{req}, L_{ret}, V, e_b\rangle \in^{c} c \\ e_b \neq \mathsf{abstract} \qquad P; E; C \vdash e_i : t_i \qquad \textbf{locksFromSet}(P, L_{req}, \mathsf{this}, c) \subseteq C \end{array}}{P; E; C \vdash_{\text{final}} \mathsf{super}.mn(e_1 \ldots e_n) : t \textbf{ as locksFromSet}(P, L_{ret}, \mathsf{this}, c)}$$

Figure 5.10: Rules for method invocation.

SYNC

$$\frac{P; E; C \vdash_{\text{final}} e_1 : c \textbf{ as } L \qquad P; E; (C \cup L) \vdash e_2 : t}{P; E; C \vdash \mathsf{synchronized}(e_1)\ \{e_2\} : t}$$

Figure 5.11: Rule for synchronized blocks.

required lock names are converted to locks of the receiver expression e. When invoking a method on the super class, the receiver is this, and the locks are looked up with respect to the super class. A method call is a final expression if the receiver expression is a final expression. Super method calls are always final expressions because the receiver can be named by this. A final method invocation represents the lock, if any, declared to be returned by the method with respect to the receiver object.

The rule for the synchronized block is shown in Figure 5.11. This rule is the ultimate consumer of the expression-as-lock information. The lock expression e_1 is required to be a final expression, and the set of locks it must represent are added to the lock context used to evaluate the body of the block, e_2.

Rules for the remaining expressions are shown in Figure 5.12. These expressions do not require checking any lock-related properties, nor do they directly result in locks. The fork expression evaluates its body in an empty lock context: this is because new threads do not hold any locks.

$$\textsc{Sub}\quad \frac{P; E; C \vdash e : c' \qquad c' \leq^{\mathrm{c}} c}{P; E; C \vdash e : c}$$

$$\textsc{Null}\quad \frac{P \vdash c}{P; E; C \vdash \mathsf{null} : c}$$

$$\textsc{New}\quad \frac{P \vdash c \qquad \mathbf{NoAbstractMethods}(P, c)}{P; E; C \vdash \mathsf{new}\ c : c}$$

$$\textsc{Assign}\quad \frac{P; E; C \vdash e : t \qquad E = E_1, t\, x, E_2}{P; E; C \vdash x = e : t}$$

$$\textsc{If}\quad \frac{P; E; C \vdash e_1 : \mathsf{boolean} \qquad P; E; C \vdash e_2 : t \qquad P; E; C \vdash e_3 : t}{P; E; C \vdash \mathsf{if}(e_1)\,\{e_2\}\ \mathsf{else}\ \{e_3\} : t}$$

$$\textsc{Let}\quad \frac{mod\_t = [\mathsf{final}]_{\mathrm{opt}}\ t \qquad x \notin E \qquad P; E; C \vdash e_1 : t \qquad P; E, mod\_t\ x; C \vdash e_2 : t'}{P; E; C \vdash \mathsf{let}\ mod\_t\ x = e_1\ \mathsf{in}\ \{e_2\} : t'}$$

$$\textsc{Fork}\quad \frac{P; E; \emptyset \vdash e : t}{P; E; C \vdash \mathsf{fork}\ \{e\} : \mathsf{int}}$$

$$\textsc{Seq}\quad \frac{P; E; C \vdash e_1 : t_1 \qquad P; E; C \vdash e_2 : t_2}{P; E; C \vdash e_1 ; e_2 : t_2}$$

$$\textsc{Abstract}\quad \frac{P \vdash t}{P; E; C \vdash \mathsf{abstract} : t}$$

$$\textsc{Label}\quad \frac{P; E; C \vdash e : t \qquad \mathcal{E}_P \supseteq \{(lbl, E)\}}{P; E; C \vdash [e]_{lbl} : t}$$

$$\textsc{LabelFinalExpr}\quad \frac{P; E; C \vdash_{\mathrm{final}} e : t\ \mathbf{as}\ L \qquad \mathcal{E}_P \supseteq \{(lbl, E)\}}{P; E; C \vdash_{\mathrm{final}} [e]_{lbl} : t\ \mathbf{as}\ L}$$

Figure 5.12: Rules for expressions that are neutral on locks.

Chapter 6

Concurrency Policy

The concurrency-related models, and the annotations expressing them, discussed thus far have been directly concerned with the mutable state of a program and the locks used to protect it. These models are sufficient for describing a large portion of the concurrency-related design intent, particularly in programs that do not make sophisticated use of concurrency. These models are, however, insufficient for describing important concurrency-related design intent concerning which methods of a class may execute concurrently while nonetheless maintaining the unstated representation invariants of the class. We introduce the notion of *concurrency policy* to capture this design intent. The root causes of the problems in the `BufferedInputStream` example of Section 1.4 are (1) the failure of the implementors of the class to specify a concurrency policy, and (2) the failure of the implementors of clients to respect the unstated concurrency policy.

Let us consider an additional example to further motivate the expression of concurrency policy. Figure 6.1 shows four versions of the class `EventQueue`, differing only in their use of synchronization. The difference in lock acquisitions *does* matter, and it affects the value computed by `getSize`. Indeed, the whole point is to affect the result of `getSize`. The extra lock represented by the field `gsLock` determines whether `getSize` accounts for the addition or removal of a priority event that may be occurring concurrently with the execution of `getSize`.[1] The class in Figure 6.1d is the simplest of the four—it does not use the lock `gsLock`. In particular, as a result the methods `enqueuePriority` and `dequeuePriority` can execute after `getSize` has read from `numHigh` but before it has computed the total number of items in the queue by summing the number of priority and non-priority elements. The other three versions of the class are as follows:

[1]We acknowledge the contrived nature of this example, particularly with the respect to the extent of lock acquisitions, but it is useful for explaining the problem that concurrency policy is meant to solve.

- Figure 6.1a implements an “upper bound” `getSize`, in which `enqueuePriority` is not allowed to execute concurrently with `getSize`. The computed size is maximized because priority elements are allowed to be removed from the queue after the number of priority elements, stored in `numHigh`, is read, but no priority items are allowed to be added to the queue.
- Figure 6.1b implements a “lower bound” `getSize`, in which `dequeuePriority` is not allowed to execute concurrently with `getSize`. The computed size is minimized because priority elements are allowed to be added to the queue after the number of priority elements is read, but no priority items are allowed to be removed from the queue.
- Figure 6.1c implements an “exact” `getSize`, in which neither `enqueuePriority` or `dequeuePriority` are allowed to execute concurrently with `getSize`. Thus no changes to the number of priority elements in the queue are allowed while the size of the queue is being calculated.

All four versions of `EventQueue` acquire the lock represented by the field `high` before accessing the fields `numHigh` and `high`. Similarly, all four versions acquire the lock represented by the field `normal` before accessing the fields `numNormal` and `normal` (usages of this field other than as a lock are elided). We assert that all these usages are consistent with an unshown locking model. The only real difference between the four classes is in their use of the lock represented by field `gsLock`, a lock that is not associated with any state. Without this lock, it is not possible to obtain the different behaviors in method `getSize`. The point is that associations between locks and state are not always sufficient to account for the invariants that must be maintained by a class. In particular, the internal state of the queue is not corruptible in any of the four versions, but the meaning of the `getSize` operation varies among them, and certain interleavings may be acceptable with respect to internal state and *intended* client meaning.

To see this problem at a lower level of granularity, consider that models of lock–state association allow method `dequeuePriority` to be implemented as shown in Figure 6.2. This implementation always accesses state according to the locking model but is clearly “broken” because the contents of the `Vector` referenced by `high` cannot be allowed to change after checking that the size of the `Vector` is nonzero. Indeed, a program written such that every `synchronized` block accessed exactly one protected field would be consistent with its locking model, but in practice full of race conditions. This is because the point of acquiring locks is to protect representation invariants. Why then the long-standing practice of associating locks with state? Locks are associated with sets

```
public class EventQueue { ...
  public int getSize() {
    synchronized( gsLock ) {
      int s1, s2;
      synchronized( high ) {
        s1 = numHigh;
      }
      synchronized( normal ) {
        s2 = numNormal;
      }
      return s1 + s2;
    }
  }

  public void
  enqueuePriority( Object e ) {
    synchronized( gsLock ) {
      synchronized( high ) {
        high.add( e );
        numHigh += 1;
      }
    }
  }

  private Object dequeuePriority(){
    Object e = null;
    synchronized( high ) {
      if( numHigh != 0 ) {
        e = high.remove( 0 );
        numHigh -= 1;
      }
      return e;
    }
  }
}
```

(a)

```
public class EventQueue { ...
  public int getSize() {
    synchronized( gsLock ) {
      int s1, s2;
      synchronized( high ) {
        s1 = numHigh;
      }
      synchronized( normal ) {
        s2 = numNormal;
      }
      return s1 + s2;
    }
  }

  public void
  enqueuePriority( Object e ) {
    synchronized( high ) {
      high.add( e );
      numHigh += 1;
    }
  }

  private EQEvent dequeuePriority(){
    Object e = null;
    synchronized( gsLock ) {
      synchronized( high ) {
        if( numHigh != 0 ) {
          e = high.remove( 0 );
          numHigh -= 1;
        }
        return e;
      }
    }
  }
}
```

(b)

```
public class EventQueue { ...
  public int getSize() {
    synchronized( gsLock ) {
      int s1, s2;
      synchronized( high ) {
        s1 = numHigh;
      }
      synchronized( normal ) {
        s2 = numNormal;
      }
      return s1 + s2;
    }
  }

  public void
  enqueuePriority( Object e ) {
    synchronized( gsLock ) {
      synchronized( high ) {
        high.add( e );
        numHigh += 1;
      }
    }
  }

  private EQEvent dequeuePriority(){
    Object e = null;
    synchronized( gsLock ) {
      synchronized( high ) {
        if( numHigh != 0 ) {
          e = high.remove( 0 );
          numHigh -= 1;
        }
        return e;
      }
    }
  }
}
```

(c)

```
public class EventQueue { ...
  public int getSize() {
    int s1, s2;
    synchronized( high ) {
      s1 = numHigh;
    }
    synchronized( normal ) {
      s2 = numNormal;
    }
    return s1 + s2;
  }

  public void
  enqueuePriority( Object e ) {
    synchronized( high ) {
      high.add( e );
      numHigh += 1;
    }
  }

  private EQEvent dequeuePriority(){
    Object e = null;
    synchronized( high ) {
      if( numHigh != 0 ) {
        e = high.remove( 0 );
        numHigh -= 1;
      }
      return e;
    }
  }
}
```

(d)

Figure 6.1: Four versions of `EventQueue` that differ only in the number and scope of critical sections.

```
1  private EQEvent dequeuePriority(){
2    Object e = null;
3    int size;
4    synchronized( high ) { size = numHigh; }
5    if( size != 0 ) {
6      synchronized( high ) { e = high.remove( 0 ); }
7      synchronized( high ) { numHigh -= 1; }
8    }
9    return e;
10 }
```

Figure 6.2: An implementation of `dequeuePriority` that follows the locking model of its class but that obviously has race conditions.

of fields, *regions* in our locking model, because it is convenient: most of the time the invariants associated with the set of fields *can* be preserved by simply acquiring the appropriate lock before accessing the region.

6.1 Observing State Changes

The missing design intent that would prevent `dequeuePriority` from being implemented as in Figure 6.2 is information on which portions of code are allowed to observe changes to shared state. When explicit representation invariants are available, this information is potentially deducible from the code, formal pre- and postconditions, and the invariants. Portions of code that temporarily invalidate invariants should not interleave with other portions of code that access the state whose invariants are invalidated. Returning to class `EventQueue` and method `dequeuePriority`, an obvious invariant, stated informally, is that "`numHigh` should always equal the number of elements in the vector referenced by `high`." Clearly, then, no other thread should be allowed to access `high` or `numHigh` between the removal of an element on line 6 and the decrement of the element count on line 7.

A fundamental premise of this work, however, is that it is generally unreasonable to require programmers to explicitly express representation invariants. Programming tools based on representation invariants such as ESC/Java [FLL$^+$02] appear to suffer from poor adoptability, to the extent that invariant inference engines such as Houdini [FJL01, FL01] have been developed to make the code annotation process less painful. We introduce the notion of *concurrency policy* as a surrogate for unstated representation invariants that still allows the programmer to differentiate between "good" and "bad" concurrency. We believe that programmers can specify concurrency policy without first having to formally express the representation invariants for a class; rather, concurrency policy can be developed based on informal code-focused arguments based on intuitive notions of what will "break" the code. Examples of such arguments are given in Sections 6.3.1 and 6.5.1.

6.2 Expressing Concurrency Policy

The concurrency policy of a class implementation specifies which segments of code methods have potential executions that can be safely interleaved. We choose to focus on interleaved executions because it is fundamentally this interleaving that allows changes to shared state to be observed across threads. In the simplest formulation, concurrency policy is expressed as a symmetric boolean matrix indexed by method names. For larger classes, there is a potential combinatorial challenge which can be mediated using several possible techniques. These include (1) indexing the matrix by regions or effects and deriving a method-indexed 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 revisited in Section 6.5.

Concurrency policy is useful not only to the implementor of a class, but also to the clients of a class. For example, is the client allowed to invoke methods `close` and `read` of class `Buffered-InputStream` concurrently? We thus distinguish between two uses of concurrency policy: internal policy that restricts the *implementation* of a class, and external policy that restricts the *use* of a particular implementation.[2] These uses are described in more detail in the following sections.

6.3 Internal Concurrency Policy

The *internal concurrency policy* of a class sets the upper bound on the extent of interleaving for the methods of a class and its subclasses. In other words, the internal concurrency policy is an approximation of the representation invariants that define safe *vs*. unsafe concurrency for a class implementation. Its focus is on the maintenance of the integrity of the shared state by the implementation and specifically *not* on the client-side correct use of the class. Internal policy expresses constraints

[2]In an earlier presentation [GS02], we refer to these uses as *guiding* and *client* policy, respectively.

but not the mechanisms used to prevent interleaving, which may be resource- or client-side. An implementation could use any of the following mechanisms, 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 method implementations may have both safe and unsafe interleavings. A method-level boolean formulation of internal policy cannot capture this distinction, and thus two methods must not be allowed to interleave if *any* of their interleavings is unsafe.

The internal concurrency policy for a class is captured in a separate design document linked with the class it describes. An implementation can be checked to verify that the internal 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 explicit representation invariants, internal concurrency policy is a design decision and so must be trusted. While techniques such as those of [OG76, Lam80] may help verify the appropriateness of an internal policy with respect to explicit representation invariants, our experience in multiple case studies is that informal reasoning (such as performed below) is often sufficient. Tools can help with this process, for example, by presenting to the programmer the different interleavings that must be considered.

6.3.1 An Example

We now consider as an example the internal concurrency policy for the class `BoundedFIFO`, shown in Figure 6.3. 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 "S" for `InfoMethods` × `InfoMethods` means that any pair in the Cartesian product is safe, where *safe* means that their concurrent execution preserves the state's integrity. Some of the informal reasoning based on the implementation of `BoundedFIFO` used to develop this policy is as follows:

- Method `get` cannot be allowed to interleave with itself because it could result in both calls to `get` returning the same element. Referring to Figure 1.2, this could happen if a second thread began to execute `get` after the first thread executed through line 15.
- Method `put` cannot be allowed to interleave with itself because it could result in two events being stored in the same location in the underlying array, causing one of the events to be lost.

	`get`	`put`	*`InfoMethods`*
`get`	×		
`put`	×	×	
`InfoMethods`	S	S	S

`InfoMethods` = { `getMaxSize`, `length`, `wasEmpty`, `wasFull`, `isFull` }

Figure 6.3: The internal concurrency policy for `BoundedFIFO`. An "S" indicates allowable safe, *i.e.*, invariant-preserving, interleaving; "×" that interleaving is unsafe and disallowed.

This could happen if a second thread began to execute `put` after the first thread had executed through line 24.

- Methods `get` and `put` can be allowed to interleave with each other. The predicate `first == next` is only true when the FIFO is full or empty, in which cases `put` or `get`, respectively, will execute without accessing the contents of the array. Thus it can never be the case that two threads try to access the same element of the array referenced by `buf`. In practice this concurrency is hard to exploit because of the synchronization required to access the shared state and the exclusions required to prevent `get` and `put` from interleaving with themselves. Because of this difficulty, we make the design decision that the internal policy should prevent the interleaving.
- The `InfoMethods` set can clearly be allowed to interleave with itself because all the methods only read state.

There are other "bad" results that can happen as a result of these interleavings, but these observations are sufficient to prevent them as well as those explicitly identified.

6.3.2 Internal Policy and Implementation

The reasoning used to develop the internal concurrency policy is necessarily informed by the implementation of the class, and, as we see in Section 6.5, by the external concurrency policy of referenced classes. The purpose of the internal policy is to describe what interleavings the *implementation* of the class, where implementation includes constraints placed on clients of the class, must not allow to be possible. If any one of the interleavings is allowed by the implementation, then it would be possible for the representation invariants of the class to be violated. This implies that the internal policy must actually be developed with respect to some potentially non–thread-safe version of the class. This is why the above reasoning uses a version of `BoundedFIFO` that does not contain lock annotations. An internal policy developed with respect to the protected version of the class, as in Figure 5.4, would in fact be fully permissive because all of the methods are already prevented from interleaving with each other. Section 8.3.3 demonstrates the development of a non-trivial internal policy from code that contains region-based critical sections.

It is sometimes necessary to introduce a new lock for the purpose of enforcing policy decisions. The lock represented by `gsLock` in Figure 6.1 is an example of such a lock. We call a lock used for this purpose a *policy lock* and do *not* associate it with a region. The programmer may document the intent that a lock is a policy lock using the class annotation

`@policyLock` *lockName* `is` *representation*

This is basically the same as a `@lock` annotation but without the region association. Currently there are no policy-related analyses that take direct advantage of this annotation. The lock analyses described in Section 5.6 use this annotation to support lock identification. This prevents “unknown lock” warnings from being generated when a `synchronized` block acquiring a policy lock is encountered. Section 7.6 gives an example of using a `@policyLock` annotation.

6.4 External 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 internal policy as long as clients can be assured not to exploit it. The *external 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

`@safeWith` $method_1$, ..., $method_n$

declares that methods m, $method_1$, ..., $method_n$ may be invoked concurrently by clients. The name of a method set may be used in a `@safeWith` annotation to refer to all the methods in the set. To be compatible with inheritance, method pairs not mentioned in `@safeWith` annotations 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 `@safeWith` inherited methods and may also redeclare as safe method pairs previously asserted to be interfering.

The external policy thus describes to clients of a class which potentially unsafe method interactions they must avoid. That is, if two methods are not `@safeWith` each other, then a client that allows them to be invoked concurrently could cause the representation invariants of the class to be violated. Pairs of methods that are `@safeWith` each other are only guaranteed to not cause the class's representation invariants to be violated. They specifically are *not* guaranteed to actually execute interleaved; that is they may execute serialized even if they are invoked concurrently. For example, a class implemented as a monitor takes full responsibility for protecting itself and thus has an external policy declaring all pairs of methods as safe. External 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 external policy clearly must not be more permissive than the internal 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.

6.4.1 External Policy and Object Composition

For the purposes of analyzing a single client, the external policy alone defines the contract for assuring correct usage—there is no need to examine the internal policy or other annotations on the resource. In the absence of overall guidance on how the external policy is to be enforced, however, 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. Locking preconditions, via `@requiresLock` annotations, help document design intent of how the external policy is to be met. If an object whose client is responsible for enforcing policy is uniquely referenced by its client, then the client may use any technique to enforce the object's concurrency policy because no other clients can exist.

As an example, we consider the case of `BoundedFIFO` and a client class `BlockingFIFO`, based on the canonical clients of Figure 5.2, that wraps a `BoundedFIFO` instance to produce a blocking FIFO implementation. Figures 6.4 shows `BoundedFIFO` annotated with external policy annotations, among others; `BlockingFIFO` annotated with external policy is shown in Fig-

```
/**
 * @methodSet InfoMethods = getMaxSize, length, wasEmpty, wasFull, isFull
 * @methodSet calleeLocked = get, put, InfoMethods
 * @lock BufLock is this protects Instance
 * @set InfoMethods reads Instance
 * @set calleeLocked requiresLock BufLock
 * @set calleeLocked safeWith InfoMethods */
public class BoundedFIFO { ...
  /** Returns <code>null</code> if empty.
   * @writes Instance */
  public LoggingEvent get() { ... }

  /** If full, then the event is silently dropped.
   * @writes Instance */
  public void put(LoggingEvent o) { ... }

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

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

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

  /** Returns <code>true</code> if the buffer was full
   *  before the last get operation. */
  public boolean wasFull() { ... }

  /** Is the buffer full? */
  public boolean isFull() { ... }
}
```

Figure 6.4: Class `BoundedFIFO` with external concurrency policy annotations.

ure 6.5. The convention of `BoundedFIFO` is that clients are responsible for coordinating their use of `BoundedFIFO` instances: the external policy of `BoundedFIFO` is therefore identical to its internal policy. In addition, the `@requiresLock` annotations on its methods describe the mechanism through which clients should coordinate their actions. This insures, for example, that the canonical clients of Figure 5.2 both use the same coordination technique, avoiding the problem mentioned above.

The external policy for `BlockingFIFO`, on the other hand, is fully permissive. This reflects the implementation, in which synchronization on `fifo`—in compliance with the external policy and locking annotations of `BoundedFIFO`—already prevents the proscribed interleavings of *both* `BoundedFIFO` and `BlockingFIFO`. Class `BlockingFIFO` guarantees this to its own clients through its own external policy, expressed via `@safeWith` annotations. We note, however, that while `length` is safe with `put` and `get`, the sensible use of `length` in conjunction with either `get` or `put` is in the domain of the concurrency policies of the clients of `BlockingFIFO`.

```
/** @methodSet WrappedFIFO = put, get, length */
public class BlockingFIFO {
  private final BoundedFIFO fifo;

  public BlockingFIFO(int size) { fifo = new BoundedFIFO(size); }

  /** @safeWith WrappedFIFO */
  public void put(LoggingEvent e) {
    synchronized(fifo) {
      while(fifo.isFull()) {
        try { fifo.wait(); } catch(InterruptedException ie) { }
      }
      fifo.put(e);
      if(fifo.wasEmpty()) fifo.notify();
    }
  }

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

  /** @safeWith WrappedFIFO */
  public int length() {
    synchronized(fifo) { return fifo.length(); }
  }
}
```

Figure 6.5: `BlockingFIFO` class as a client of `BoundedFIFO`.

6.4.2 Self-Protected Objects

It is convenient to declare that objects of a particular class take sole responsibility for protecting their invariants. Such objects are traditionally what one expects when thinking of a "thread-safe" object. They have the advantage that clients of the object do not need to do anything special before invoking methods on them. The class annotation

```
@selfProtected
```

documents the design intent that a class, and its subclasses, take full responsibility for protecting themselves. It is like a fully permissive external policy, *i.e.*, every method is safe with every other method, except that it additionally requires that any method added by a subclass is also safe with all other methods.

We emphasize again that the techniques used by a class to be compliant with its declared external policy are unspecified. In general, we envision this annotation to be used on classes that implement either (1) immutable objects or (2) monitors. The analyses that would be used to verify policy compliance in these two case are different, and the `@selfProtected` annotation by itself does not provide enough information to know which technique is being used. It has the advantage, however, of being abstract, and thus, for example, an interface can be annotated as being `@selfProtected` and have multiple implementations, some of which are immutable and some of which are monitors.

In Parameterized RACEFREEJAVA [BR01], a class may be "self-synchronized." Such classes implement objects that always own themselves, which in the idiom of their system means that instances of the class must always lock on themselves to protect their state. This is more specific than what our `@selfProtected` annotation is declaring; a self-synchronized class is one way in which a `@selfProtected` class could be implemented.

6.5 A Policy Mismatch Error

We now give an example from Log4j version 1.1b5 in which the external policy of a provider class is not followed by the implementations of its clients. This "policy mismatch" creates several sources of null-pointer and index-out-of-bounds exceptions, as well as enabling a violation of an internal object protocol [Gre01]. These exceptions can cause the program being logged to terminate prematurely or portions of the logging functionality to misbehave.

The Log4j library is introduced in Section 1.3; here we describe additional details relevant to this example. Logging-event sources implement the `AppenderAttachable` interface, shown in Figure 6.6; `Appender` objects are chained together by attaching to classes that implement `AppenderAttachable`. The library provides two classes that implement the interface. To avoid duplicating the implementation, the core functionality of managing event listeners is implemented by the class `AppenderAttachableImpl`, partially shown in Figure 6.7. This class does not make use of synchronization. The `AppenderAttachable` interface is implemented by classes `Category` and `AsyncAppender`, neither shown, whose instances each have their own unaliased `AppenderAttachableImpl` instance to which they delegate calls.

Unlike the case of `BoundedFIFO`, which also originates from the Log4j library, there is no discernible convention on client synchronization, even though it is clearly the responsibility of the clients of `AppenderAttachableImpl` to protect it from concurrent access. In `Category` and `AsyncAppender`, the methods `addAppender`, `removeAppender(String)`, `removeAppender(Appender)`, and `removeAllAppenders` are `synchronized`. The uses of method `appendLoopOnAppenders`, a method which is not part of the `AppenderAttachable` interface, are synchronized within `Category`, but not in `AsyncAppender`. This synchronization is not sufficient to

```java
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 6.6: Log4j's `AppenderAttachable` interface.

```java
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 6.7: A portion of the implementation of Log4j's `AppenderAttachableImpl` class.

	Instance
Instance	×

(a)

	read Instance	write Instance
read Instance	S	
write Instance	×	×

(b)

Figure 6.8: (a) Monitor-like and (b) reader–writer external policies for AppenderAttachable-Impl expressed using regions and effects, respectively. “S” indicates concurrent execution is safe; “×” indicates concurrent execution is prohibited.

prevent null-pointer and index-out-of-bounds exceptions during concurrent use, as will become apparent when we examine the external concurrency policy we inferred for AppenderAttachable-Impl. First we infer a correct external policy, then we illustrate how it is violated, and finally we suggest how analyses might detect the policy violations.

6.5.1 External Policies for AppenderAttachableImpl

Generally speaking, the most conservative external policy is a policy that does not allow the client to execute any pair of methods concurrently. This is most concisely recorded using the region-based policy shown in Figure 6.8a: 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 description, shown in Figure 6.8b, differentiating methods based on their effects on an 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 the effects of the class implementation.

Returning to our example, the most liberal external concurrency policy for AppenderAttachableImpl is shown in Figure 6.9. This policy is identical to the class’s internal 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 their only possible interactions are through the Vector referenced by appenderList, and the external policy of Vector, were the JDK appropriately annotated, indicates that any one of its methods is @safeWith any other. An analysis informed by the effects of AppenderAttachableImpl and the external policy of Vector could deduce the safety of these methods. The other five safe inter-

		(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 6.9: The most liberal external concurrency policy for `AppenderAttachableImpl`. A bold "S" indicates differences from the reader–writer policy. Pairs marked * are safe due to the underlying `Vector`.

leavings 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 6.7.

- We do not declare `addAppender` to be `@safeWith` `addAppender`. 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.

- We declare `addAppender` to be `@safeWith` `appendLoopOnAppenders`. In developing this policy we consider two interactions. (1) `addAppender` can assign a new `Vector` to `appenderList` if there is no existing 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 external policy.

- We do not declare the `Appender`-taking version of `removeAppender` to be `@safeWith` `appendLoopOnAppenders`. If `removeAppender` removes an element from the list after `appendLoopOnAppenders` has read the size of the list, then an index-out-of-bounds exception can result because `appendLoopOnAppenders` will iterate past the end of the list of appenders. This is an example of how even though the methods of a class, in this case `Vector`, may be `@safeWith` each other, their concurrent use must still be coordinated at a higher level to preserve higher-level invariants.

- We declare `removeAppender(Appender)` to be `@safeWith` itself because the the concur-

		(1)	(2)	(3)	(4)	(5)	(6)	(7)
(1)	`addAppender`	$\times$						
(2)	`appendLoopOnAppender`	S	S					
(3)	`getAllAppenders`	S	S	S				
(4)	`getAppender`	S	S	S	S			
(5)	`removeAllAppenders`	$\times$	$\boxed{\text{S}}$	$\boxed{\text{S}}$	$\boxed{\text{S}}$	$\times$		
(6)	`removeAppender(Appender)`	$\boxed{\times}$	$\boxed{\text{S}}$	S	$\boxed{\text{S}}$	$\times$	$\boxed{\times}$	
(7)	`removeAppender(String)`	$\boxed{\times}$	$\boxed{\text{S}}$	S	$\boxed{\text{S}}$	$\times$	$\times$	$\times$

Figure 6.10: The policy enforced by clients of `AppenderAttachableImpl`. Boxes indicate where the enforced policy is different from the external policy.

rency policy for `Vector` indicates that `removeElement` is `@safeWith` itself.

6.5.2 Detecting the Mismatch

Failure to use enough synchronization within `Category` and `AsyncAppender` enables seven interleavings disallowed by our inferred policy and could thus cause unwanted exceptions. The policy enforced by these two classes in their usage of an instance of `AppenderAttachableImpl` is shown in Figure 6.10. Differences from the external policy are boxed; in particular, the seven occurrences of an $\boxed{\text{S}}$ indicate errors in the implementations of the clients. A static analysis can detect the policy violations if the following annotations are added (1) an external policy for `AppenderAttachable-Impl`, and (2) uniqueness of the delegate references in `Category` and `AsyncAppender`. Clearly the external policy must be specified before it can be enforced. Because `AppenderAttachable-Impl` has no locking annotations, it does not provide any hints regarding how clients should enforce its external policy. By annotating that the delegate references are `@unshared`, we insure that those instances will have exactly one client, and thus the client is free to protect the delegate in any manner it chooses. In this particular case, both `Category` and `AsyncAppender` lock themselves to protect the delegate.

A static analysis based on extant locking to enforce the external policy would proceed by identifying all the locks that *must* be held at each site where a method on the `AppenderAttachable-Impl` instance is called. Let m_i be the set of locks that must be held at call site i of method m. Then for each pair of methods (m, n) that the external policy does not declare as safe, such that m has call sites drawn from set p and n has call sites drawn from set q, the client correctly enforces the policy pair if $\left(\bigcap_{i \in p} m_i\right) \cap \left(\bigcap_{j \in q} n_j\right) \neq \emptyset$. That is, there exists a lock that must always be held whenever m is called and whenever n is called.

We omit the details of how the policy mismatch can be fixed; see [Gre01]. We do note, however that it is nontrivial to implement a general-purpose client that is not more conservative than the policy in Figure 6.9. There is likely to be little performance benefit, because the overhead of synchronization is likely to outweigh the benefits of concurrency. The benefits of such a liberal external policy, however, stem from the *documentation of design intent*. By documenting external policy, clients are given more information about how the class is intended to be used. For example, were `BufferedInputStream` given a policy that indicated that `read` and `close` were not safe to invoke concurrently, the whole scenario described in Section 1.4 would be irrelevant.

6.6 Policy Representation Revisited

The method-based approach to expressing policy has several problems. An obvious problem is that it does not scale well. For classes with more than a handful of methods, expressing the policy is likely to be burdensome. Considering the method interactions is more burdensome still. The combinatorics of policy expression can be reduced by instead expressing policy in terms of interactions over regions, as briefly explored in Section 6.5. There is an obvious trade-off between conciseness of expression and flexibility in expressed policies.

A second problem with method-based policy expression, that can also be addressed by switching to a region-based policy, is scoping the concurrency policy. 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 `@safeWith` various mutator methods of the class, but the resulting enumeration object is *not* because it is an unprotected access path to the underlying state of the `Vector`. 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.

6.7 Related Work

We believe that our presentation of concurrency policy is a novel approach to the specification and maintenance of unstated representation invariants. In particular, none of the tools discussed in previous chapters, *i.e.*, ESC/Java [FLL+02], RACEFREEJAVA [FF00], Parameterized RACEFREEJAVA

[BR01], and Guava [BST00], support the description of concurrency policy.

Noble, Holmes and Potter describe an "algebra of exclusion" for reasoning about exclusion in composite objects [NHP00]. The underlying semantics of their algebra is identical to that of our method-based concurrency policy: pairs of methods whose executions must exclude each other. Their algebra represents the matrix symbolically. A difference is that they are not concerned with relating the matrix to implementation, rather they are concerned with relating the matrix to the architecture of a system of objects. They envision that an implementation is derived from unsynchronized code and a specification of the exclusion constraints via an aspect-oriented programming system. They are primarily concerned with being able to reason about two things: (1) that the exclusion constraints of a composite object satisfy those of its delegate objects, and that therefore the aggregated objects do not require exclusion in their implementation; and (2) that the exclusion implemented by delegate objects is sufficient to satisfy the constraints of the composite object. Because they are not focused on implementation, there is no assurance that implementations actually implement the exclusions, or that aggregated delegate objects are actually properly contained to satisfy the assumptions of their logic. In this regard, their work complements ours: while our work is concerned with the assurance that an implementation is consistent with its policy, their work is concerned with assuring that the interaction of the policies of cooperating objects produces the desired effects.

Several languages contain non–lock-based syntactic constructs for implementing mutual exclusion that superficially resemble concurrency policy. In Path Pascal [CH74], class-level path expressions specify both object protocol and concurrency constraints. In a language proposed by Andrews and McGraw [AM77] each method is annotated with those methods that may execute concurrently with it (a.k.a. compatibilities). CEiffel [Löh93], a multi-threaded dialect of Eiffel, uses a similar annotation. Subclasses may, however, 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 effects "to give the programmer some control over the way in which computations are interleaved" [Luc87]. All monitor-call effects are defined to interfere with each other. Described as an external 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 [SS84]. 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.

Flanagan and Qadeer have developed a type system for specifying and checking atomicity properties [FQ03b] and have applied it to Java [FQ03a]. This system is an extension of the race-free type system used by RACEFREEJAVA, and is meant to solve the problem that race-freedom as defined by their previous work does not account for the interleaving of critical sections, as discussed in Section 6.1. Their type system is based upon Lipton's theory of reduction [Lip75], and categorizes expressions based on their ability to interleave with the actions of other threads. Programmers must type a method with its intended atomicity. Such a type system could be adapted for assuring that a class implementation is consistent with its internal concurrency policy.

Chapter 7

Tools

We have implemented a prototype tool that embodies the analysis techniques described in previous chapters. Our experience with the tool provides some preliminary evidence of the practicability of our approach for ordinary programmers on deadlines. As mentioned throughout, our design decisions are fundamentally influenced by the desire that our approach be adoptable by working programmers. This influences our approach to recording design intent. Specifically, we avoid recording explicit representation invariants: we record intent using annotations that describe models of mechanical program behavior. These annotations are intended to answer questions that the programmer is already thinking about. The design of the tool is also influenced by our focus on adoptability. We wish to provide the programmer using our tool with "early gratification"—some assurance should be obtained with minimal or no annotation effort, and additional increments of annotation should be rewarded with additional increments of assurance or warnings of model–code inconsistencies. Our prototype tool thus supports an interactive and iterative assurance process. The programmer builds up assurance results by gradually executing assurance analyses and introducing annotations. Thus for our principle of early gratification to be satisfied, a programmer must be rewarded with useful analysis results within the first few iterations, and should not have to introduce large amounts of annotations.

The design of our tool allows assurance to proceed incrementally across the code base and across the models of intent. Unlike similar tools, such as RaceFreeJava and Guava, we do not require that the entire program to be assured thread-safe at once, a fundamental obstacle to providing early gratification. These tools use modular type systems, but the assumption is that the whole program is being made thread-safe at once. This is evident, for example, by the requirement that all fields be associated with a lock, or the assumption that any reference to an object is going to be a safe object. Such assumptions make it difficult to obtain meaningful incremental results because there is no provision for distinguishing code that is not yet annotated from code that is annotated and inconsistent with its annotations. These approaches force the programmer to annotate everything about every class before meaningful results are possible. Our approach, on the other hand, is to consider annotations as specifications of models with which source code should be consistent. Additional annotations thus describe additional models of design intent. An unannotated class is merely a class that has no models that it should be assured against. Assurance of an annotated class may be compromised by the absence of annotation on another class; we must be careful, therefore, to place failure appropriately on the lack of design information on the unannotated class, as well as not to be report misleading results. Our tool thus reports a class of messages, "warnings," that do not indicate code–model inconsistency, but rather call to the attention of the programmer places where additional assurance is probably possible if additional design intent is specified.

An additional distinguishing feature of our approach, and its embodiment in a prototype tool, is that we wish to go beyond "bug hunting." It is just as important to provide positive assurance that source code is consistent with intent as it is to report the inconsistencies. Similar tools such as ESC/Java are intended to statically find errors that would otherwise only be caught at runtime, if at all, because their manifestation may be nondeterministic. Such tools, which include model checkers, therefore, are only interested in *negative* results, and are typically incapable of providing explicit *positive* assurance that the program is consistent with the specified model of design intent. A run with no warnings must usually be interpreted to mean that the tool was unable to find any inconsistencies, but cannot reliably be interpreted to mean that the model is satisfied by the program. Our tool should only be silent when there are no models are assure. Future work is to provide "chains of evidence" that link together expressed design intent, analysis results, and code segments to document the reasoning behind each assurance result.

In what follows, we first describe our prototype assurance tool and how it is incorporated into the Eclipse IDE. We then briefly discuss our framework for representing data in general and for representing Java programs specifically. This is followed by a discussion of the annotations and analyses supported by our prototype tool and some notes on their implementation. This chapter concludes with two examples of using our prototype tool to assure the safety of classes taken from production Java code.

7.1 The Fluid Eclipse Plug-Ins

Our prototype assurance tool is implemented as a set of plug-ins to the open source Eclipse[1] Integrated Development Environment (IDE).[2] Integration with an existing IDE allows our engineering effort to be focused on the implementation of our analysis and assurance infrastructure rather than the mundane tasks of maintaining a Java front-end. Incorporation within an IDE is also important pedagogically because of our desire that the programmer have an incremental and iterative "conversation" with the assurance tool: to be adoptable, our tool must integrate well with existing tools used by a typical programmer.

Figure 7.1 shows an example of an Eclipse workspace window containing our tool, visible as the "Code Quality Advice" view below the environment's standard Java editor. Here, the editor shows a portion of a partially annotated version of the `BoundedFIFO` class. Our assurance tool has been run on `BoundedFIFO.java` and the rest of the "BoundedFIFO Example" project (visible in the "Package Explorer" view on the left). The results are shown sorted by issue in the "Code Quality View;" they may also be sorted by file. The presentation of assurance results is a fundamental issue in the adoptability of our tool and is on-going research. Results in our "Code Quality View" include:

- Positive assurance of model–source code consistency. These results are marked by an icon representing an unbroken chain. Specifically, analyses found 11 call sites where methods annotated with lock preconditions were called with those preconditions satisfied, and 15 field accesses where the fields are accessed with the appropriate lock held.
- Identification of model–source code inconsistencies, *i.e.*, assurance failures. These results are marked by a broken chain icon. Here we see that analysis is unable to assure that nine field accesses are consistent with the annotated locking model. Expanding the top-level node in the view reveals the specific accesses that may violate the model. As expected, double-clicking on these results focuses the editor window on the offending line of source code.
- Warnings of missing intent—also (unfortunately) marked by a broken chain icon. In this case, our tool warns that there are two references to a possible shared unprotected object; see Section 5.7. This is distinguished from an assurance failure because no annotated model is actually being violated.

Our prototype assurance tool is implemented as two Eclipse plug-ins. The *Fluid Plug-in* is

[1] `http://www.eclipse.org/`

[2] The design and implementation of our prototype tool is an on-going effort by all members of the Fluid group.

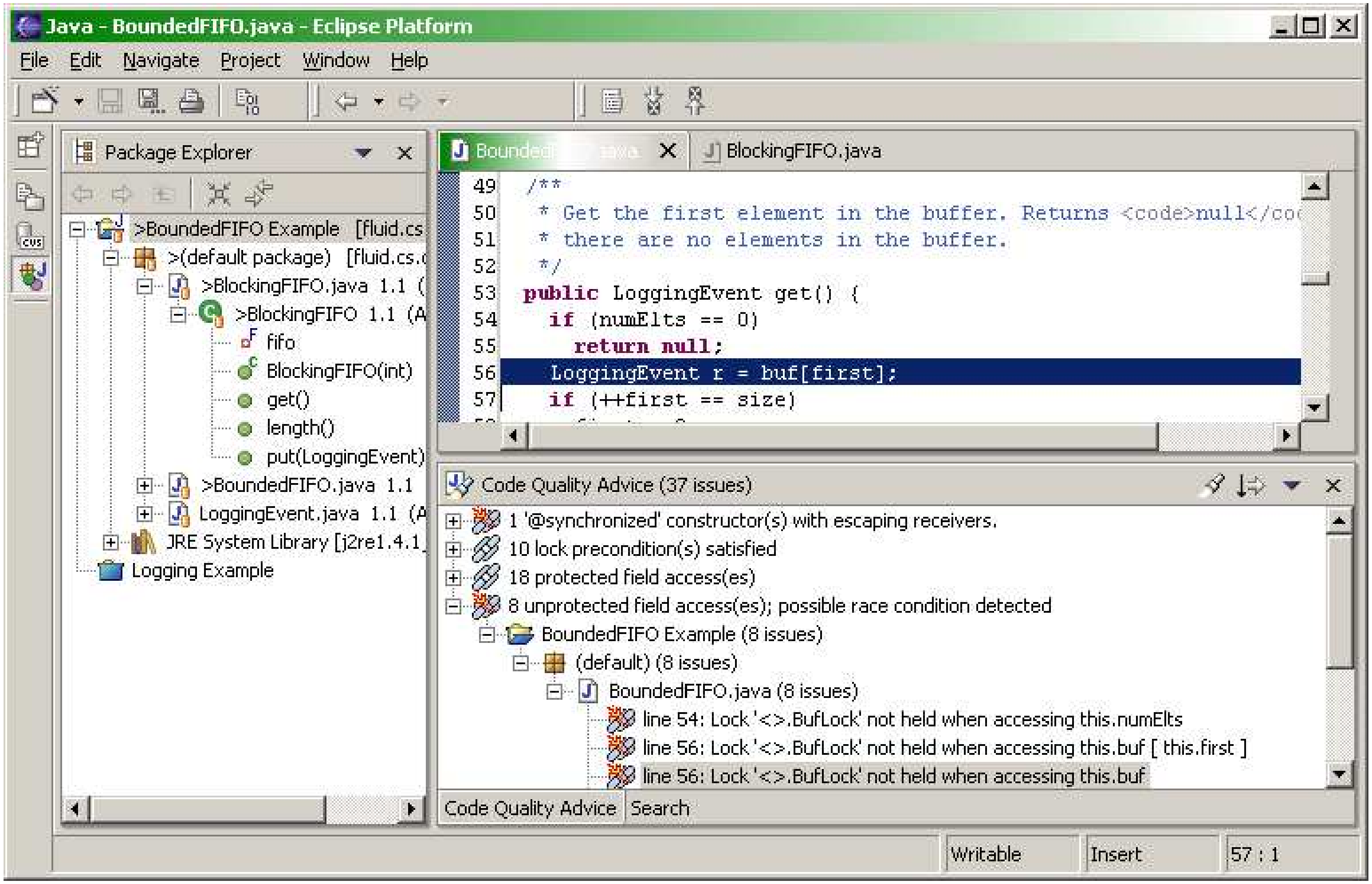


Figure 7.1: The Eclipse IDE with our prototype assurance plug-in. Our plug-in provides the “Code Quality Advice” view.

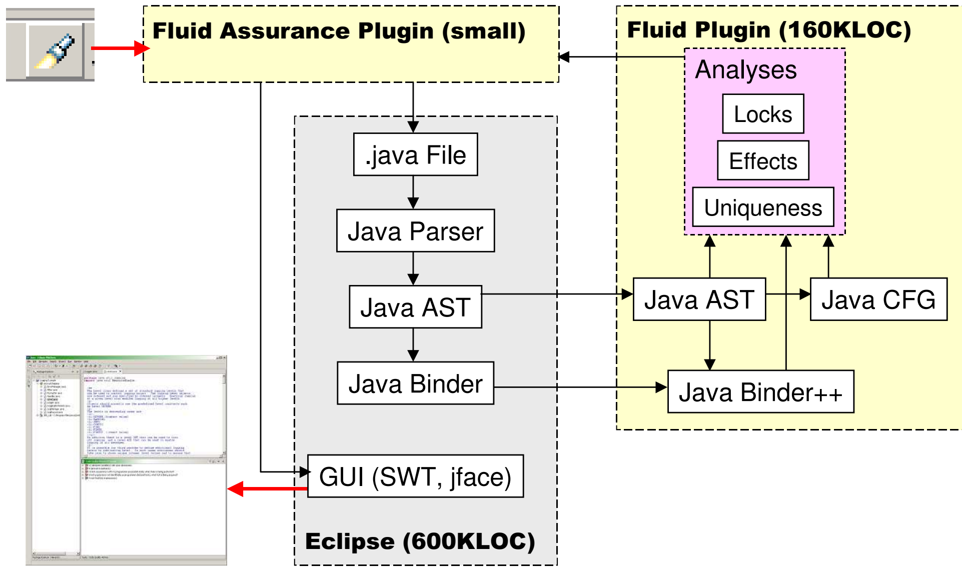


Figure 7.2: Data flow in our prototype assurance tool.

a library of our analyses, data structures, and other concerns that are not handled by Eclipse. It also contains the routines for converting from Eclipse representations to our own. This plug-in is approximately 160,000 lines of Java code (160kLOC). It does not define any Eclipse views or other user-visible GUI items. The *Fluid Assurance Plug-in* defines the "Code Quality View" and provides the means for an Eclipse user to run assurance analyses and interact with the results. It is relatively small, only several thousand lines of Java code. Eclipse itself is approximately 600kLOC. Figure 7.2 shows the general relationship between the two plug-ins and the Eclipse environment.

The programmer, *i.e.*, the Eclipse user, interacts with our tool via the "Code Quality View" provided by the Fluid Assurance Plug-in. The programmer runs the assurance analyses by clicking on the flashlight icon in the view's toolbar. This causes all the Java source files in all the open projects to be analyzed. This "analyze everything" approach is an artifact of boot-strapping the tool implementation. We emphasize that our analyses *are not* interprocedural or whole-program: each class is analyzed independently using only the annotated interfaces of other classes. Annotations serve as cutpoints, enabling global analysis to be avoided and opaque components to be integrated safely, contingent on the validity of their annotations. Eclipse allows analyses to be run automatically as source code is changed; indeed, this is one of the reasons we chose to integrate our tool with Eclipse. Using this framework to drive our analyses is on-going work and should enhance the usability and adoptability of our tool.

More specifically, the Fluid Assurance Plug-in bridges the functionality of Eclipse and the Fluid Plug-in and provides our tool's user interface. The Fluid Assurance Plug-in drives analysis by querying Eclipse about the workspace and obtaining resource handles to all the Java files in the workspace. For each Java file, the plug-in has Eclipse parse it into an abstract syntax tree (AST). From this AST, a parallel AST in our own representation is created; Section 7.2 discusses the rationale for this. Each analysis, the code for which resides in the Fluid plug-in, is invoked on this parallel AST. The implementation of our analyses and their interdependencies are discussed in more detail in Section 7.4.1. Analysis results, *i.e.*, positive assurances, *etc*., are reported back to the Fluid Assurance Plug-in via callbacks.

Several of our analyses are data-flow–based and require control-flow graphs (CFGs). Our Fluid Plug-in provides an infrastructure for building CFGs from our AST representation; see Section 7.2.2. Analysis also requires binding information, that is, the binding of lexical names to their definitions in the AST. Eclipse provides a basic binder, which we wrap in the Fluid Plug-in (1) to insure that binding results are relative to our AST instead of the the Eclipse-based AST, and (2) to provide additional information not directly available from the Eclipse binder.

The Fluid Assurance Plug-in gathers analysis results via callbacks and displays the results from all the analyses in the "Code Quality View" as a tree. Currently, the results may be displayed sorted by issue, as seen in Figure 7.1, and by package and class. The programmer may browse the results; double-clicking on a particular issue—positive or negative—focuses the Eclipse editor on the relevant line of code.

7.2 Representations

Our plug-ins use many special purpose program representations in parallel to those provided by the Eclipse API. These data structures are built on top of a data representation framework that we call the Fluid Internal Representation, or IR. This framework predates our work with the Eclipse IDE and provides sophisticated versioning and persistence features, that while unused by the current prototype tool, are critical for the project's longer term goals of studying tools that manage program evolution. Here we first briefly describe the IR framework, and then describe the program representations built on top of it.

7.2.1 The IR

The IR models general purpose data using a modified version of the standard ternary representation: unique identifiers, attributes, and values. The modifications we make include (1) ultra-fine grained tree-structured versioning, (2) abstraction to structured entities such as trees and directed graphs, and (3) persistence. These are also several other features not used in this prototype.

Our implementation represents the unique identifiers as objects that implement a distinguished interface, `IRNode`, that defines a small set of operations for getting and comparing identity, and for getting and setting the value associated with that identifier and a specific attribute. Attributes are simply containers for values indexed by unique identifiers, and may optionally be named and typed. A type in this case refers to an IR data type, not the Java notion of type. IR types are represented by objects implementing a special interface, which includes methods for determining if an object should be considered an instance of the type and for persisting members of the type. The framework provides a set of primitive types including the standard scalar types, *e.g.*, integer, string, *etc.*, as well as several complex types such as sequences and records. Unfortunately, using this kind of meta-level type system means that programming for the IR involves an abundance of type casts and, in general, is unable to take advantage of the type safety provided by the Java programming language. (The absence of generic type types in Java contributes to this problem.) Attributes are represented by objects that are maps from unique identifiers to values. Any unique identifier may be given a value for any attribute. Attributes and identifiers can be created dynamically.

Complex structures are built in the IR by considering a given set of identifiers and a given set of attributes to define the scope of the data structure. For example, trees can be represented by mapping each node in the tree to a unique identifier, by defining "parent" and "child" attributes, and by managing the values of those attributes as appropriate to maintain the structure of the tree. Maintaining representation invariants over such a diverse collection of objects is difficult. The IR, therefore, contains classes that provide familiar high-level APIs, implementations of which manage and abstract the underlying attributes. For example, the `Tree` class provides expected tree methods, such as `getParent`, `getChildren`, `setChild`, and `depthFirstSearch`; these methods manage the "parent" and "child" attributes "under the hood." Specifically, the IR provides a sophisticated library of graph classes built on top of the identifier–attribute abstraction.

Data in the IR actually has a third component: version. Most simply, a complete identifier–attribute–version triple represents a particular value at particular point in time. Any change to any value produces a new version. The IR maintains a pointer to the current version, which may be queried and reset to a previous version. Our implementation is transparently optimized by not remembering any versions that are never asked for. An additional feature of IR versioning is that the version space is tree-structured rather than "time-ordered" linear. A tree-structured version space allows for experimentation, by providing the ability to return to a previous version and making new changes to the values, which starts a new branch in the version tree. The IR provides a shadow model of the version space implemented as an IR tree data structure.

Finally the IR supports persistence, that is, the storage and restoration of *subsets* of in-memory IR structures to and from the file system. The intention is that these subsets be defined along the lines of components and other higher-level abstractions. Persistence preserves identity, in particular if a particular identifier is persisted into two different (but overlapping) subsets of data, then even if the two subsets are loaded into two different JVMs the identifier is preserved in both JVMs. Identity is even preserved through structures such as Java `ObjectStreams`. The result is the implementation of the abstraction that all identifiers always exist.

7.2.2 Representing Java Programs

Our tool represents a Java program using an augmented abstract syntax tree (AST). These trees are stored in the IR. They are represented as IR trees containing an additional "operator" attribute. In addition to describing the role of the node in the program, *e.g.*, "if statement," "field reference expression," or "method declaration," the value of this attribute constrains the number of children the node may have and dictates what their operators may be. It may appear to be redundant to build our own AST when Eclipse already maintains one. We are motivated to use our own AST for several reasons. From the point of view of expediency, many of our analyses were written prior to our use of Eclipse and are based on our AST representation. More to the point, several of our analyses are control-flow based and therefore require control-flow graphs (CFGs). Eclipse does not provide CFGs; we already have a framework for generating CFGs from our own ASTs. As a project, we made the engineering decision to write a component that translates Eclipse ASTs into Fluid ASTs so that we would not have to modify already existing and working analysis code.

Our decision to continue to use our own AST representation is additionally motivated by longer-term project goals of developing new techniques for (1) visualizing and browsing source code and models of design intent and (2) managing program evolution. An IR-based AST is necessary for both. As previously mentioned, versioning is intrinsic to the IR—but is not native to Eclipse—and thus an IR-based AST trivially supports evolution management. Our project has also developed a model–view–controller (MVC) framework on top of the IR that supports elaborate control over how models, *e.g.*, Java programs as ASTs, are displayed. Eclipse provides an elegant and extensive set of GUI components that assist us in the ultimate rendering of our models, but does not provide a suitable MVC framework. Thus supporting within Eclipse our project's visualization research still requires the translation of Eclipse ASTs into Fluid IR representations.

Our control-flow graphs are built from IR-based ASTs following a "cookie cutter" design pattern in which each node in the AST, via its operator value, defines a control-flow component, that has one or more "holes" in it filled by the node's children. More specifically, our control-flow graphs contain explicit edges and nodes and support traversals bidirectionally. Analysis results are stored on the edges. The CFG framework is not Java specific: language specific control-flow components are built out of primitive node types that abstractly determine how lattice values propagate among subcomponents. The framework provides a generic work-list–based control flow algorithm that is parameterized by a language and analysis specific transfer function that determines the specifics of how a particular operator affects the incoming lattice value.

7.3 Annotating Missing Code

For our tool to be practical it is important that it be possible to annotate "missing" source code with models of design intent. Source code may be missing, for example, because a library is only distributed in binary form, or because a class may not be implemented yet even though its interface has been designed. Even when the source code for a library is available, it is convenient to be able to separate annotations added by the user of the library from the source code of the library itself, such as in the case of the Java standard libraries. Our tool is capable, therefore, of reading annotations from external XML-formatted "promise" files. In general, the convention is that the tool looks for annotations for the class *package.class* in the file *package.class*`.promises.xml`.

Figure 7.3 shows the relevant portions of the `java.lang.Object.promises.xml` file that describe the locking preconditions for the `wait` and `notify` methods. A private region `WaitQueue` is declared on line 7. The lock `MUTEX` is declared to protect that region and to be represented by `this` on line 9. Finally, the `wait` and `notify` methods are given the annotation `@requiresLock MUTEX` on lines 17, 22, 27, 32, and 37. We also see that the constructor is annotated to have no effects, line 12, and to not create aliases to the newly created object, line 13.

ESC/Java also allows the external annotation of source code using specification files [LNS00, §5.1.2]. Such files are basically annotated Java source files except that (1) only one class may be described in the file, and (2) method bodies may be elided, and are ignored even if they are present. The `rccjava` tool for analyzing RACEFREEJAVA does not support the external annotation

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE package SYSTEM "promises.dtd">

<package name="java.lang">
  <class name="Object">
    <promise keyword="region" contents="private WaitQueue"/>
    <promise keyword="lock"
             contents="MUTEX is this protects WaitQueue"/>

    <constructor>
      <promise keyword="writes" contents="nothing"/>
      <promise keyword="borrowed" contents="this"/>
    </constructor>

    <method name="notify">
      <promise keyword="requiresLock" contents="MUTEX"/>
      <promise keyword="reads" contents="nothing"/>
    </method>

    <method name="notifyAll">
      <promise keyword="requiresLock" contents="MUTEX"/>
      <promise keyword="reads" contents="nothing"/>
    </method>

    <method name="wait">
      <promise keyword="requiresLock" contents="MUTEX"/>
      <promise keyword="reads" contents="nothing"/>
    </method>

    <method name="wait" params="long">
      <promise keyword="requiresLock" contents="MUTEX"/>
      <promise keyword="reads" contents="nothing"/>
    </method>

    <method name="wait" params="long, int">
      <promise keyword="requiresLock" contents="MUTEX"/>
      <promise keyword="reads" contents="nothing"/>
    </method>

    <!-- And so on... --->
  </class>
</package>
```

Figure 7.3: A portion of the file `java.lang.Object.promises.xml` that contains annotations for the class `java.lang.Object`. The annotations themselves are shown in boldface.

of libraries, and, in fact, specifically suppresses any warnings associated with declarations within the `java.*` packages.[3]

7.4 Annotations and Analyses

Our prototype tool implementation supports a subset of the annotations and analyses described in the previous chapters. Some of the analyses are integrated into our prototype tool, while others exist only as stand-alone demos. Table 7.1 lists all the annotations introduced herein and summarizes the extent to which they are supported by our prototype tool. An annotation that is recognized by our tool and incorporated into our internal data structures is considered "in tool." Analysis may use annotations in two ways, reflecting the use of annotations as cutpoints: (1) to assure that source code is consistent with the design intent described by the annotation, and (2) to support the assurance of another kind of design intent. In the table, the first use is considered "assured" and the second "consumed." We mark a class of annotations as consumed only if it is used by an analysis actually present in the tool. Figure 7.4 graphically shows which annotations and analyses depend on each other, fully considering effects and uniqueness. Region annotations, for example, are used not only by both lock and effects analysis, but by effect and lock annotations because they provide a vocabulary for describing state.

There are some caveats to note. (1) Our prototype tool recognizes all the previously described annotations except for the meta-level method aggregation annotations and those related to region parameters and external concurrency policy. (2) We also do not yet link the internal concurrency policy to the code. (3) Robustness is lacking, in particular, the tool does not currently enforce that the annotations be well formed, so it is possible, for example, to associate a region with multiple locks, or to extend non-existent regions. The behavior of analyses in such situations is, of course, undefined and the tool may, as result, throw exceptions during analysis.

Not all of the annotations need to be checked for consistency against the source code. The region hierarchy annotations define a model of state that influences the assurance of other pieces of design intent. Thus the `@region`, `@mapInto`, and `@aggregate` annotations have the *not applicable* marker "n/a" in the "assured" column of Table 7.1. The `@methodSet` annotation, while not recognized by our tool, also does not require any intrinsic assurance. Continuing the list of caveats, (4) we have not yet defined the model defined by `@policyLock` annotations—although it clearly must derived from the internal concurrency policy—and thus there is not yet any annotation–source

[3] Personal communication, Stephen Freund, January 2003.

Annotations		**Cutpoint Roles**	
Annotation	**In Tool**	**Assured**	**Consumed**
Method Aggregation			
@methodSet *mset* = $method_1$, ..., $method_n$ [, ...]	N	n/a	N
@set *mset annotation*	N	n/a	N
@inSet *mset*	N	n/a	N
Region Hierarchy			
@region *visibility* [static] *region* [extends *parentRegion*]	Y	n/a	Y
@mapInto *parentRegion*	Y	n/a	Y
Region Parameters			
@<region $region_1$, ..., $region_n$>	N	N	N
@<$target_1$, ..., $target_n$>	N	N	N
Effects			
@reads $target_1$, ..., $target_n$	Y	N*	N
@writes $target_1$, ..., $target_n$	Y	N*	N
Unshared References			
@unshared	Y	N*	Y
@borrowed	Y	N*	Y
@aggregate s_1 into d_1, ..., s_n into d_n	Y	n/a	Y
Locking Model			
@lock *lockName* is *representation* protects *region*	Y	Y	Y
@requiresLock $lockName_1$, ..., $lockName_n$	Y	Y	Y
@returnsLock *lockName*	Y	Y	Y
@synchronized	Y	Y	Y
Policy			
@safeWith $method_1$, ..., $method_n$	N	N	N
@policyLock *lockName* is *representation*	Y	n/a	Y
@selfProtected	Y	N	Y

Table 7.1: Annotations supported by our prototype tool. An "N*" in the assured column means that the analysis is implemented but is not currently integrated with our prototype tool.

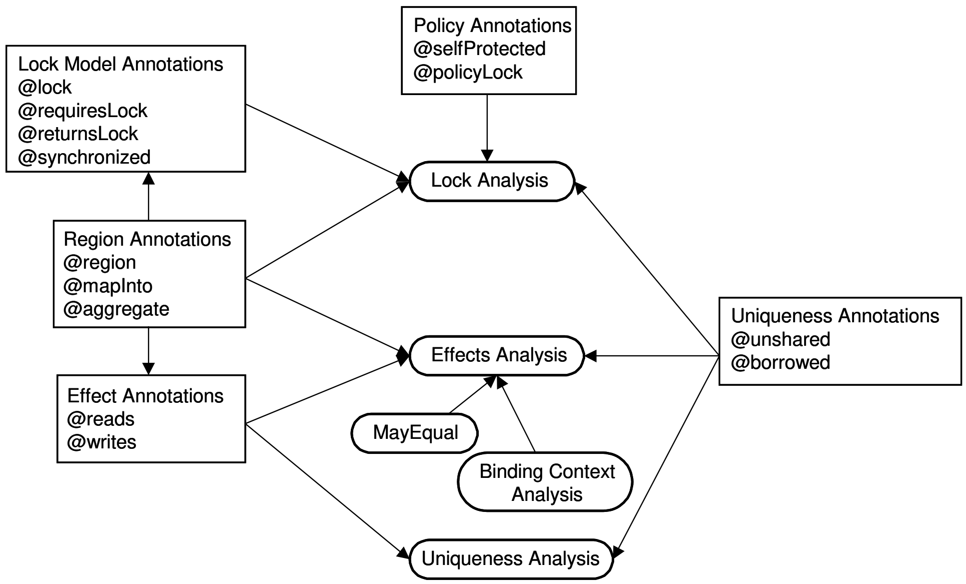


Figure 7.4: Flow of annotation and analysis information.

code consistency to assure for them either. Finally, (5) assurance analyses for effects and unshared references have been implemented, but they are not yet incorporated into our prototype assurance tool because they do not directly relate to concurrency issues.

7.4.1 Implementation Notes

Our analyses are primarily predicated on the region hierarchy and state aggregations defined by the `@region`, `@mapInto`, and `@aggregate` annotations. Our internal representations allow queries based on the tree defined by the annotations, *e.g.*, queries of region inclusion. As previously mentioned, region parameterization is not supported in our prototype tool. Above the region abstraction, we implement representations for targets, which are used extensively by the effects and locking analyses. Effects analysis is implemented, although not integrated with our prototype assurance tool. The implementation is based on the presentation of effects in [GB99], including uniqueness aggregation. Effects analysis is implemented as a depth-first walk along the syntax tree. Comparison, elaboration, and other operations are defined as operations on effect, target, and region objects. The results of comparison operations, *e.g.*, effects conflict, are designed to be helpful for providing feedback to the programmer, and thus return bit sets encoding rationale for the result, such as “the first effect is a write whose target overlaps with the target of the second effect because they are instance

Positive Assurances:	Method returns the correct lock (at method declaration) Method's lock preconditions satisfied (at call site) Field is accessed with the correct lock held (at field reference) Thread-local constructor keeps receiver from escaping (at constructor)
Assurance Failures:	Method does not return the correct lock (at method declaration) Method's lock preconditions not satisfied (at call site) Field is not accessed with the correct lock held (at field reference) Thread-local constructor may allow receiver to escape (at constructor)
Warnings:	Synchronized method does not access any shared state Synchronized block does not access any shared state Lock expression is not a final expression Cannot identify lock Field references a possible shared unprotected object

Table 7.2: The specific positive assurances, assurance failures, and warnings that our prototype tool generates.

targets whose object expressions may be aliased and whose regions overlap." Because our tool does not currently make use of effect conflict results, we have not yet incorporated an alias analysis. Our implementation of *MayEqual* is thus the most conservative possible: it always returns *true*.

Our implementation of effects uses what we call "binding context analysis," formally defined in Section 3.4. This analysis is implemented using our data-flow analysis framework and does not rely on any annotations.

Uniqueness analysis as described in [Boy01a] is implemented, although not integrated into our tool. It assures that source code is consistent with `@unshared` and `@borrowed` annotations. It is implemented using our data-flow analysis framework. There is a dependence between the uniqueness and effects analyses. The cycle of assurance is broken by using separately assured annotations as cutpoints in the assurance process. That is, effects uses uniqueness annotations trusting that they are correct, and uniqueness uses method effect annotations trusting that they are correct.

The prototype tool implements a set of lock analyses based on the annotations and analyses of Chapter 5. We present the analysis as a type system in Section 5.11, although our implementation is as a suite of separate analyses, so that, for example, it is straightforward to

- Determine the locks held when a given expression is evaluated.
- Determine the lock required to access a field. This exploits uniqueness aggregation in a

limited way to account for the aggregation of arrays.

- Determine the locks required to invoke a method.
- Determine if a method returns the correct lock.
- Determine what locks an expression may represent.

These building blocks make it easy to check whether any particular field access or method call is consistent with the locking model. In addition to the assurance analyses, we also implemented the heuristics of Section 5.7. Table 7.2 lists the specific lock-related positive assurances, assurance failures, and warnings that our prototype tool generates.

Lock analysis is a consumer not only of lock annotations but also of region annotations, uniqueness annotations, and policy annotations. The analyses are implemented as traversals over the abstract syntax tree and manipulate lock objects that encapsulate lock identity operations. As described in Section 5.6, our implementation uses “final expressions” to simplify the identification of locks. A `@synchronized` constructor is assured by verifying that it is also annotated with the design intent that it does not produce permanent aliases to the newly created object: `@borrowed this`. As previously mentioned, however, our tool does not currently assure `@borrowed` annotations.

We have not implemented any policy-related analyses. Our tool recognizes the `@policyLock` annotation which is used to assist in lock identification. The `@selfProtected` annotation is also recognized, and used by lock analysis to determine that a referenced object is not unprotected. Our prototype does not assure, however, that the annotated class is consistent with the concurrency policy embodied in this annotation.

7.5 Assuring `BoundedFIFO`

We now use the familiar `BoundedFIFO` example to demonstrate the use of our tool to provide positive assurance of thread-safety. This example demonstrates the incrementality of our approach, in which a few annotations are added at each step, and analyses provide increments of positive assurance as more annotations are introduced. In addition to exemplifying the programmer’s interaction with the code and our analysis tool, this section also brings together all the observations we have made about `BoundedFIFO` in previous chapters. In particular, assurance issues relating to lock responsibility, lock acquisition, lock assignment, and state aggregation are addressed.

This example highlights a key difference between our *assurance-based* approach and the bug-finding approaches of other similar projects. `BoundedFIFO` has the unusual characteristic of actu-

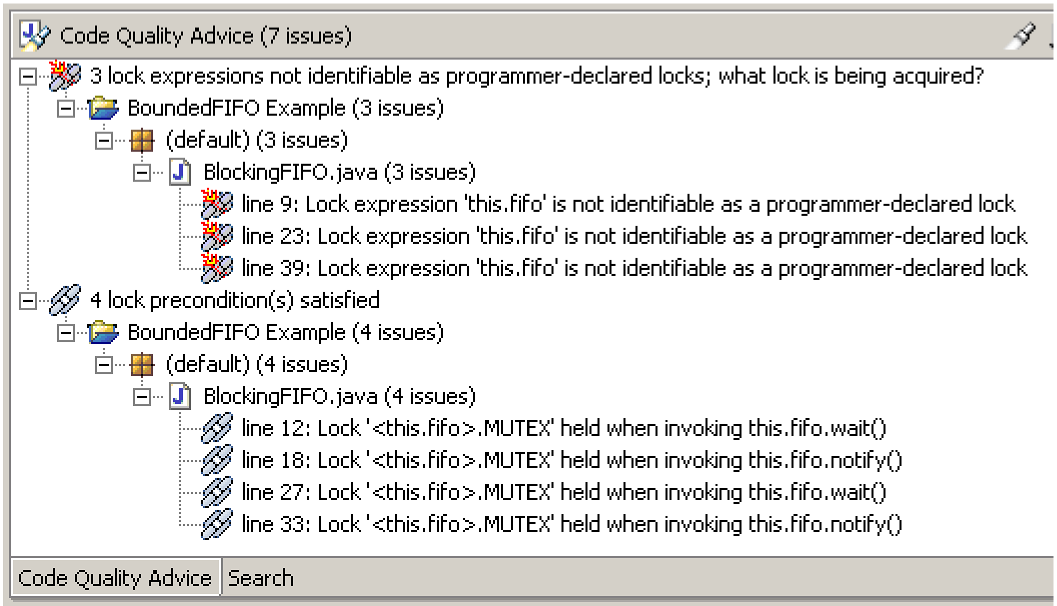


Figure 7.5: Results of applying our prototype tool to the unannotated `BoundedFIFO` source code.

ally being correctly written with respect to concurrency issues. Use of our approach provides the evidence that the program is correct. Each step of adding a new element of design intent yields additional assurances, or narrows the areas of suspicion to smaller segments of code.

Our example involves three classes (1) `BoundedFIFO` as taken from version 1.0.4 of Log4j, (2) Our wrapper class `BlockingFIFO` as a client of `BoundedFIFO`, and (3) an empty stub class `LoggingEvent` as a place holder for the Log4j class `LoggingEvent`. Initially all the classes are unannotated; `BoundedFIFO` is as in Figure 1.2 and `BlockingFIFO` is as in Figure 6.5. We will only be adding annotations to the class `BoundedFIFO`.

Running our analyses on the unannotated source code actually provides assurance because `BlockingFIFO` uses `wait` and `notify`, which are annotated with lock preconditions. Our tool reports "4 lock precondition(s) satisfied" as a result of correctly synchronizing on `fifo` before invoking `fifo.wait`, *etc.*, in `BlockingFIFO`. The results also include three warnings about "lock expressions not identifiable as programmer-declared locks; what lock is being acquired?" The output of our analyses is shown in Figure 7.5. These are in reference to the three `synchronized` blocks in `get`, `put`, and `length` respectively. No programmer-declared lock (the lock `MUTEX` declared by the Fluid system does not count) is associated with the bounded FIFO object (indeed, there are no annotations in the program yet), so analysis cannot determine what lock is being acquired by these blocks. The warning is intended to prod the programmer into introducing design intent into the program.

We annotate `BoundedFIFO` with our initial locking design intent: (1) the state is protected by locking on the object itself, and (2) the clients are expected to acquire the locks. The annotation `@lock BufLock is this protects Instance` is added to the class. Each method is annotated with `@requiresLock BufLock`. For the sake of example, we "forget" to annotate the method `get`. We rerun the analysis. The results, shown in Figure 7.6, include positive assurances, assurance failures, and warnings:

- There are now "10 lock precondition(s) satisfied." Analysis recognizes the use of `fifo` in the `synchronized` blocks of class `BlockingFIFO` as a possible use of the locks `BufLock` and `MUTEX`. Combined with the new preconditions on the `BoundedFIFO` methods, this enables the six additional method callsites to be assured. The assurance message in the results window identifies the lock expression that satisfies the precondition.
- Within class `BoundedFIFO`, the locking preconditions enable the assurance that at 15 sites, fields of the class are accessed consistently with the locking model.
- Unfortunately, analysis also reveals nine sites where fields of `BoundedFIFO` are not accessed according to the model. Here the error message indicates the lock that is expected to be held.
- There is a warning that there are 2 sites where a possibly unprotected *object* is accessed.

Inspection of the unprotected field accesses, by double-clicking on the errors in the results window, reveals that many of the unprotected accesses are in the method `get` (see Figure 7.6). This is because we "forgot" to annotate the method with a locking precondition. In general, the error could be that the implementation of the method is intended to acquire the lock. Were this the case we could satisfy the locking model by declaring the method to be `synchronized`. It is our intent, however, that the caller of the method acquire the lock, so we annotate the method with `@requiresLock BufLock`. We rerun the analyses; the results are shown in Figure 7.7. Introduction of this one annotation increases the number of assured method calls by one, because there is now one more method with a precondition to be satisfied. The number of assured field accesses increases to 22.

We now inspect the remaining two unsafe field accesses. The unsafe accesses are in the constructor of the class. As discussed in Section 5.5, it is usually safe to assume that the object is only being accessed by a single thread while it is being constructed, and thus synchronization is unnecessary. We thus annotate this design intent by adding the `@synchronized` annotation to the constructor. Rerunning the analysis provides us with additional assurances, but also with a new error; see Figure 7.8.

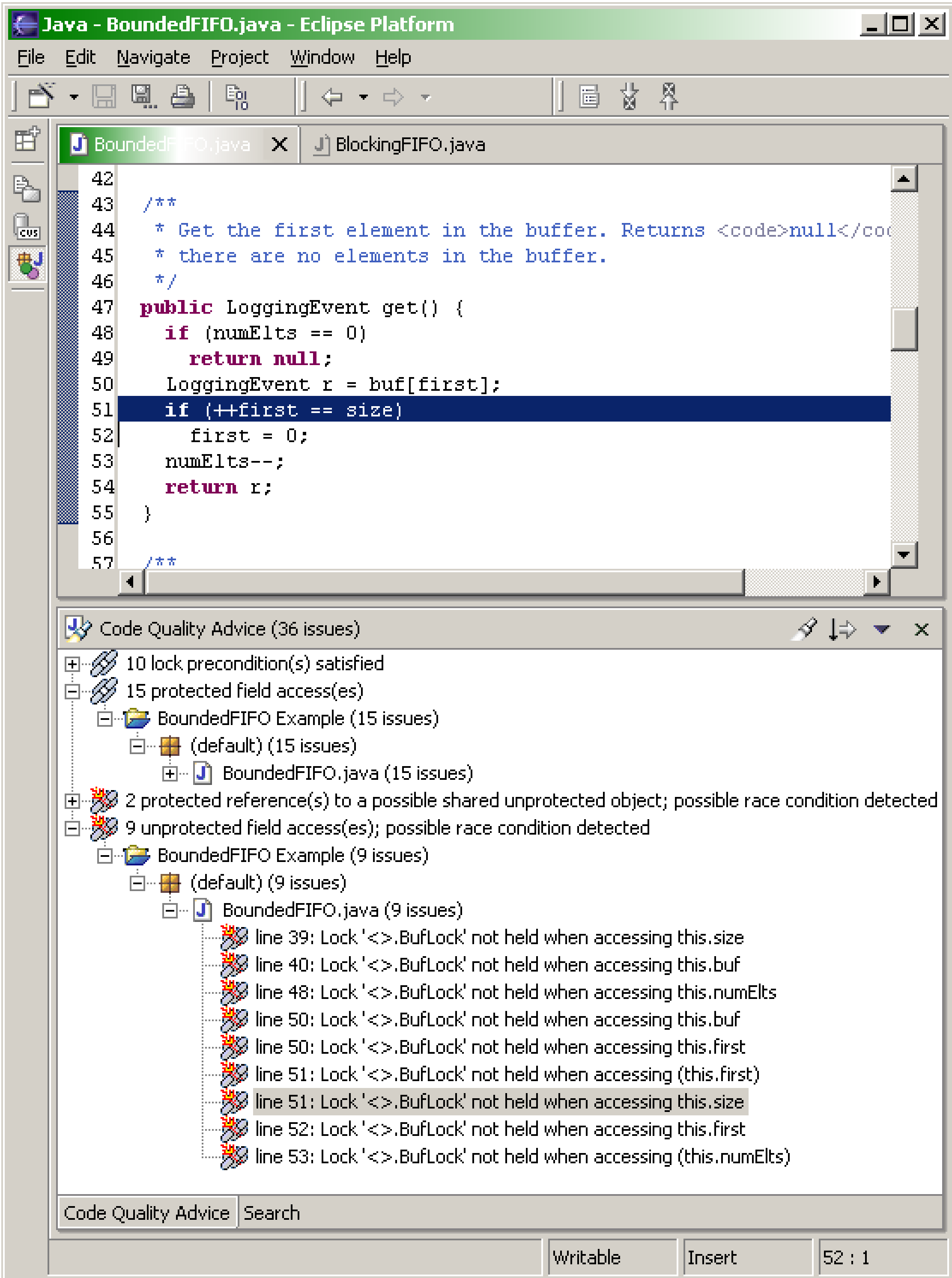


Figure 7.6: Analysis output after an initial attempt to define the locking model.

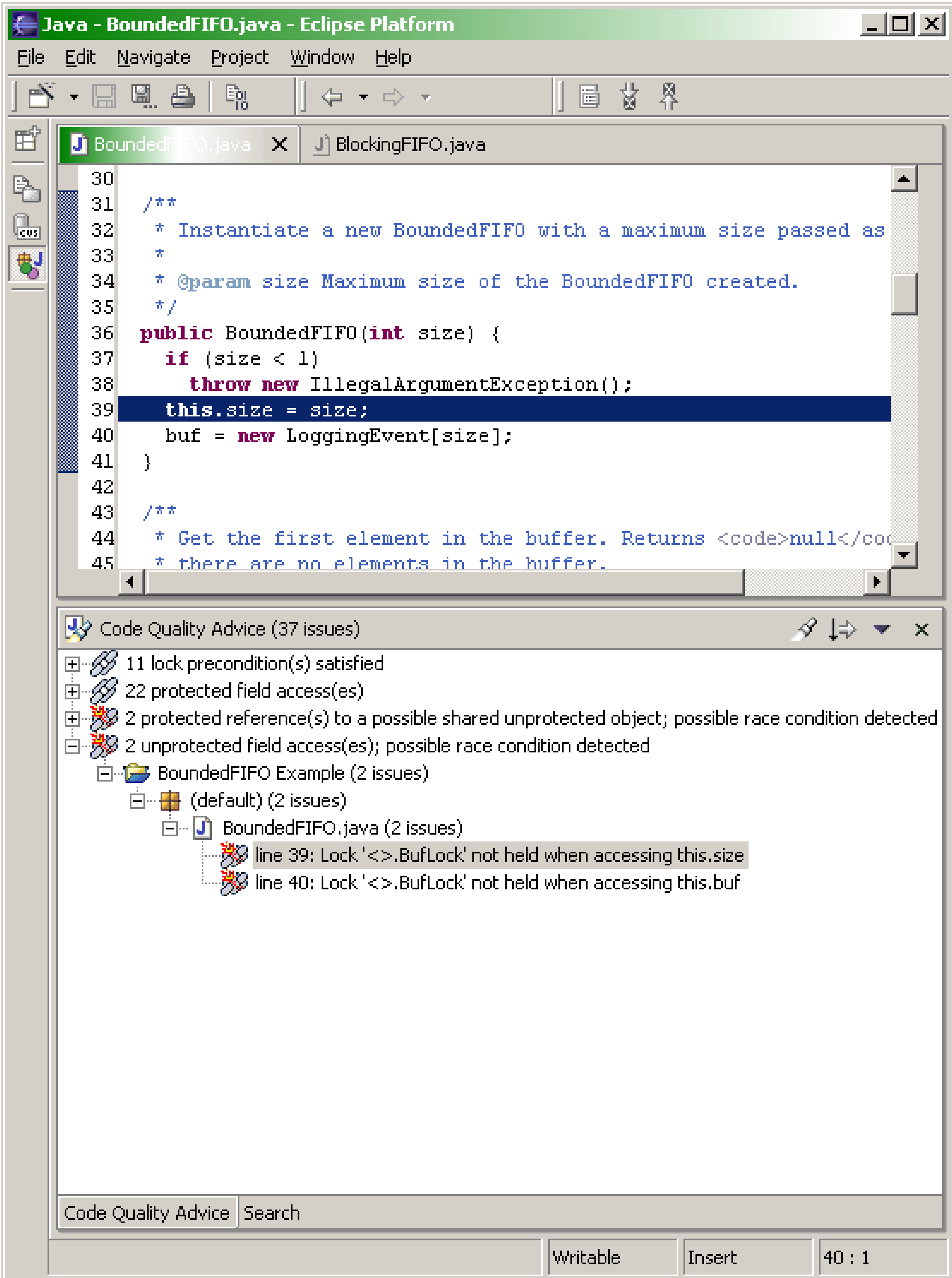


Figure 7.7: Analysis output showing “unsafe” field accesses in the constructor.

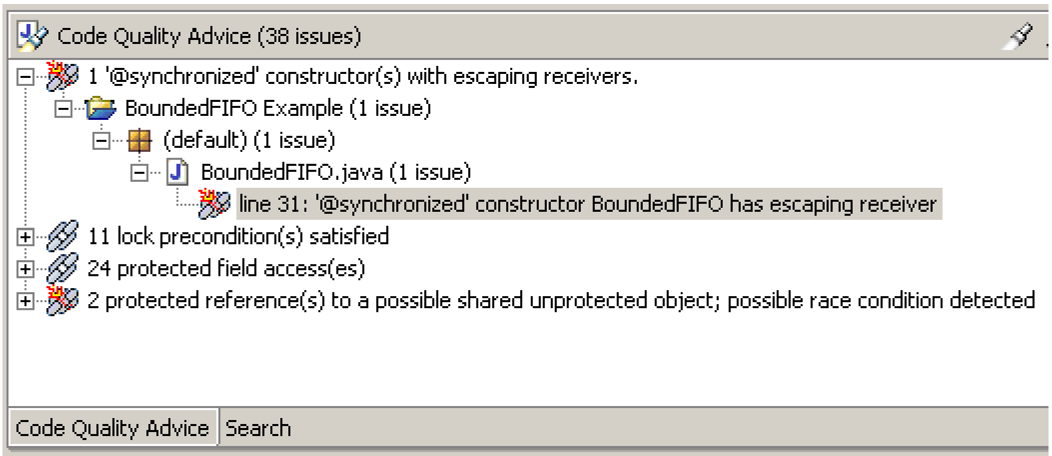


Figure 7.8: Analysis output showing the failure to assure the `@synchronized` annotation.

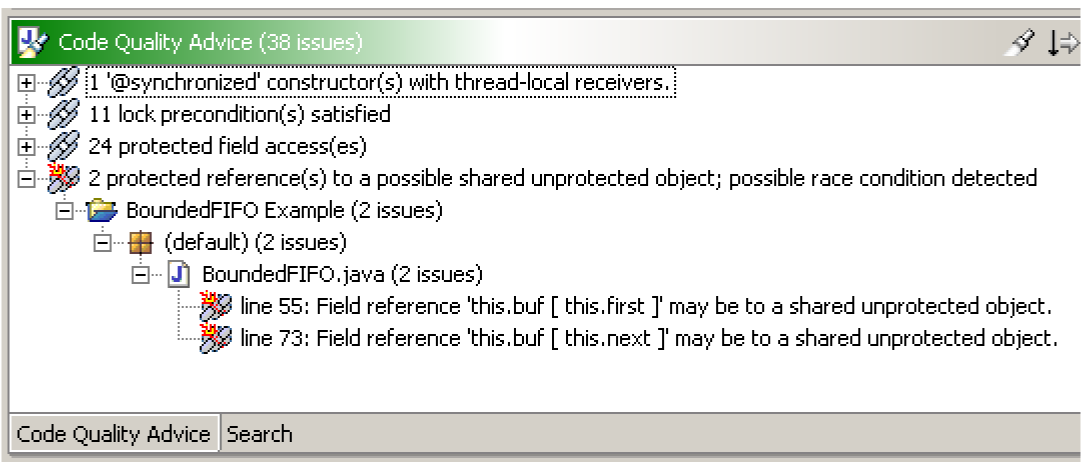


Figure 7.9: Analysis output showing possible unprotected access to the array `this.buf`.

- Documentation of the assumption allows lock analysis to assure that the field accesses in the constructor are consistent with the locking model.
- Unfortunately, analysis is not able to verify that the constructor is single-threaded, so the assurance failure “’@synchronized’ constructor BoundedFIFO has escaping receiver” is reported.

As discussed in Section 5.5, our tool implementation uses uniqueness analysis to assure the `@synchronized` annotation. We introduce the additional annotation `@borrowed this` on the constructor. Analysis is now able to assure that the constructor is indeed thread local; see Figure 7.9. (Our tool does not currently assure the `@borrowed` annotations. Here we assert that it is obvious that the constructor does not create any aliases to the newly created object.)

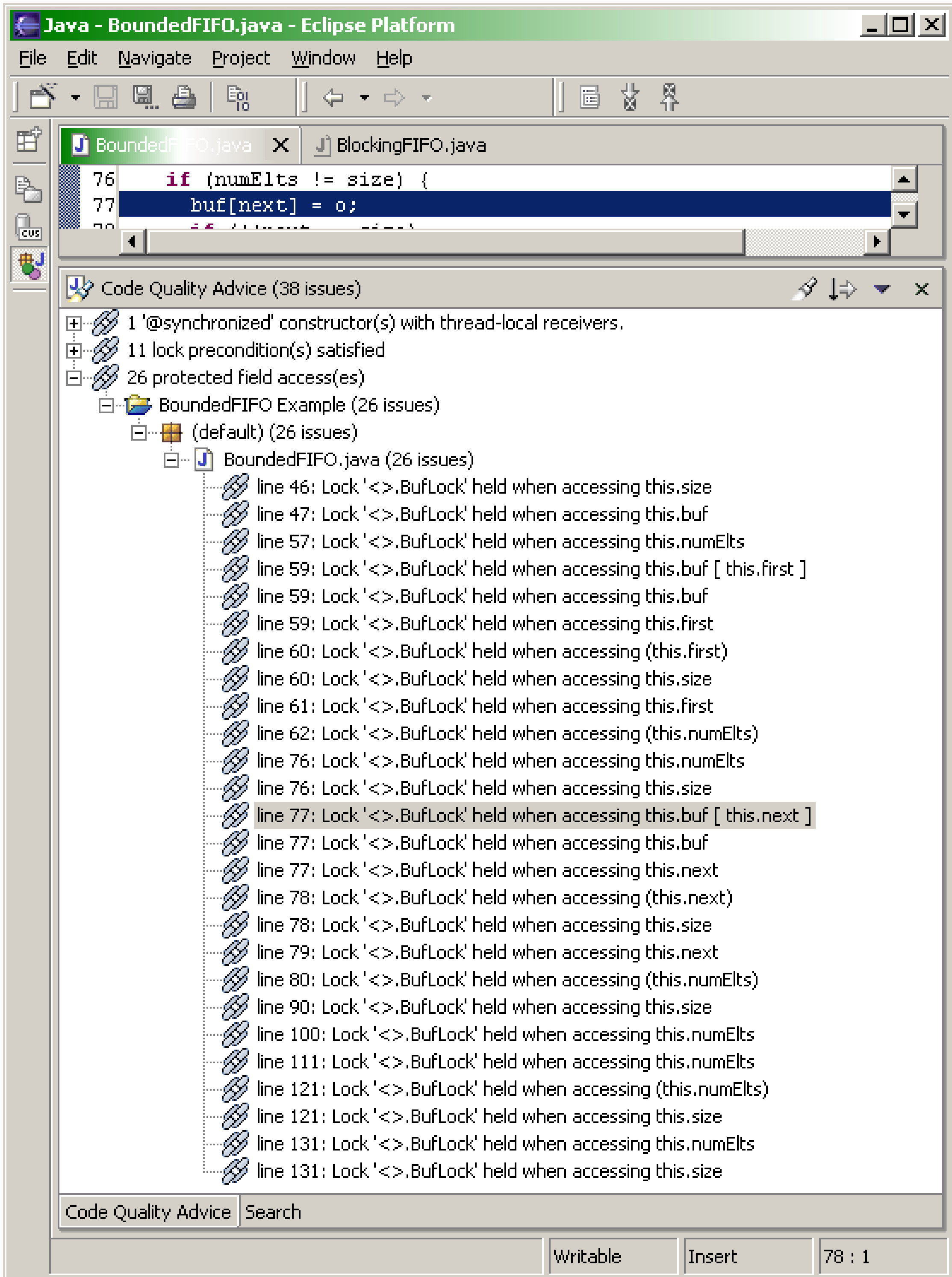


Figure 7.10: Analysis output showing full positive assurance of `BoundedFIFO` with respect it to its annotated locking model.

We now turn our attention to the remaining warning: "2 protected reference(s) to a possibly shared unprotected object." There is one warning for each dereference of the array referenced by the field `buf`; see, again, Figure 7.9. The relevant issues of design intent and assurance are discussed in other places, *e.g.*, Sections 4.6 and 5.7.1. To summarize, the tool is bringing to our attention the fact that the object referenced by `this.buf` is (1) not known to be thread-safe (that is, it does not have design intent describing how to preserve its invariants), and (2) possibly shared (that is referenced by multiple threads because the reference may be generally aliased). Thus, we may be accessing a shared object in an unsafe manner. As described previously, the solution in this case is to document the intent that the object is `@unshared` and to aggregate its state into the state of the `BoundedFIFO` object. We add the annotations `@unshared` and `aggregate [] into Instance` to the declaration of the field `buf`. One last analysis of the program now provides only positive assurances, see Figure 7.10:

- The constructor is assured to be thread-local.
- All uses of methods of `BoundedFIFO` by class `BlockingFIFO` are consistent with the preconditions of those methods.
- All uses of fields of `BoundedFIFO` are consistent with the locking model of that state, including indexing into the array referenced by `buf`.

Using our tool, we have now (1) documented concurrency-related design intent for `BoundedFIFO`, (2) associated that design intent with the code, and (3) assured that the code is consistent with that intent.

7.6 Assuring `Logger`

As a second, more complex, example of using the tool to assure production Java code, we present the class `Logger` from the Java 2 SDK, Standard Edition Version 1.4.1_01 `java.util.logging` package. Like Log4j, this package implements a logging/debugging API [Ham01]. There is one `Logger` instance for each programmer-defined component of interest. A `Logger` object thus may be shared by multiple threads, and in fact, the Javadoc for the class claims that "All methods on Logger are multi-thread safe." In this example, we must reverse engineer the original design intent. In spite of this "worst case" scenario of usage, use of our analysis tool reveals that the class contains several concurrency-related errors which we must fix before thread-safety can be assured.

We assure `Logger` in a project that contains the minimum subset of `java.util.logging` necessary to successfully compile the class. The project contains the source files for `ErrorManager`, Filter, Formatter, Handler, Level, LogManager, LogRecord, Logger, and LoggingPermission. Our analyses will report items of interest in classes other than Logger, but in this example we are only interested in assuring the single class so we do not focus on the additional errors and warnings. The class and its related files are too large to be shown here; we highlight the relevant segments of code as necessary.

This example also demonstrates our need to develop more sophisticated techniques for displaying the output of our analyses. Such concerns are beyond the scope of this dissertation, although we mention them here as a warning to the reader of the lack of useful visual structuring within the tool output shown in the figures accompanying this example.

7.6.1 A Race Condition on Field `filter`

As in the previous example, we begin by running our tool on unannotated code. Analysis reports a total of 47 issues, 21 of which are in the Logger class. Of these, 15 are related to concurrency.[4] See Figure 7.11. The first item of interest is the warning "Line 407: Lock expression 'this' is not identifiable as a programmer-declared lock." We begin our model-building process by identifying the lock represented by this. Inspection of the contents of the synchronized block reveals that the field filter is the only state of the object accessed within that critical section. Starting small—we can always modify our intent later—we annotate the class with the design intent that this is used to protect the field filter:

```
@lock L is this protects filter
```

After reanalysis, there are 19 concurrency-related issues in Logger; see Figure 7.12. Instead of the warning that this is not identifiable as a lock, analysis now returns some positive assurances, some assurance failures, and some additional warnings related to our single annotation:

- The two uses of filter in the synchronized block, both now on line 410, are positively assured to be accessed correctly.
- Two uses of filter, on line 384 (highlighted in Figure 7.12) and line 393, occur in contexts where the required lock is not held. *These are potential race conditions.*
- There are seven warnings that "Locks [<>.L, <>.MUTEX] not needed by body of synchronized method." These occur because these seven methods acquire the lock on this (by virtue of being synchronized) yet they do not use the region filter (nor do they invoke any of

[4] Our prototype tool also performs several other "code quality" analyses unrelated to those presented in this dissertation. Further discussion of these analyses is beyond the scope of this work.

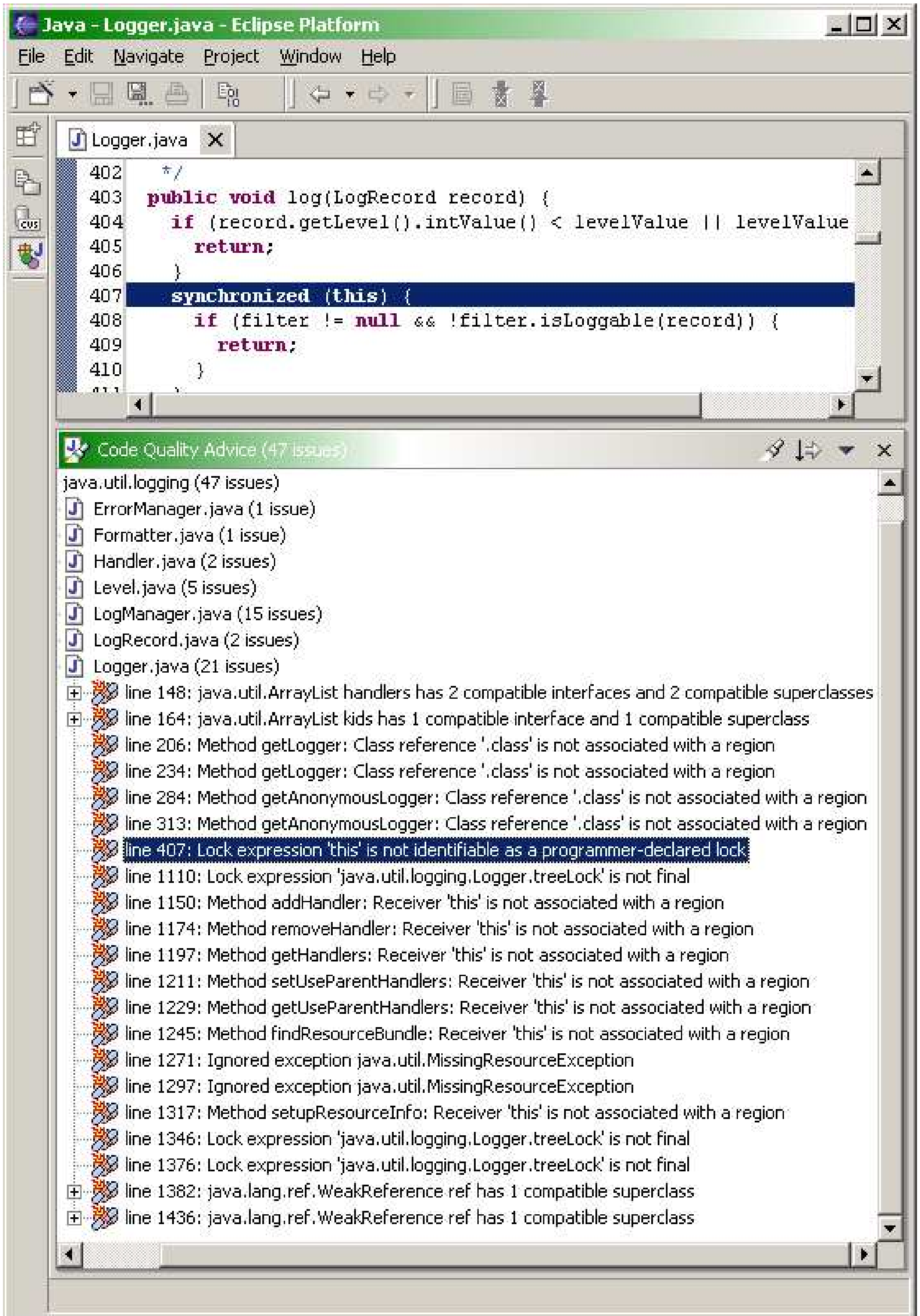


Figure 7.11: Analysis results for assuring `Logger` with no annotated model.

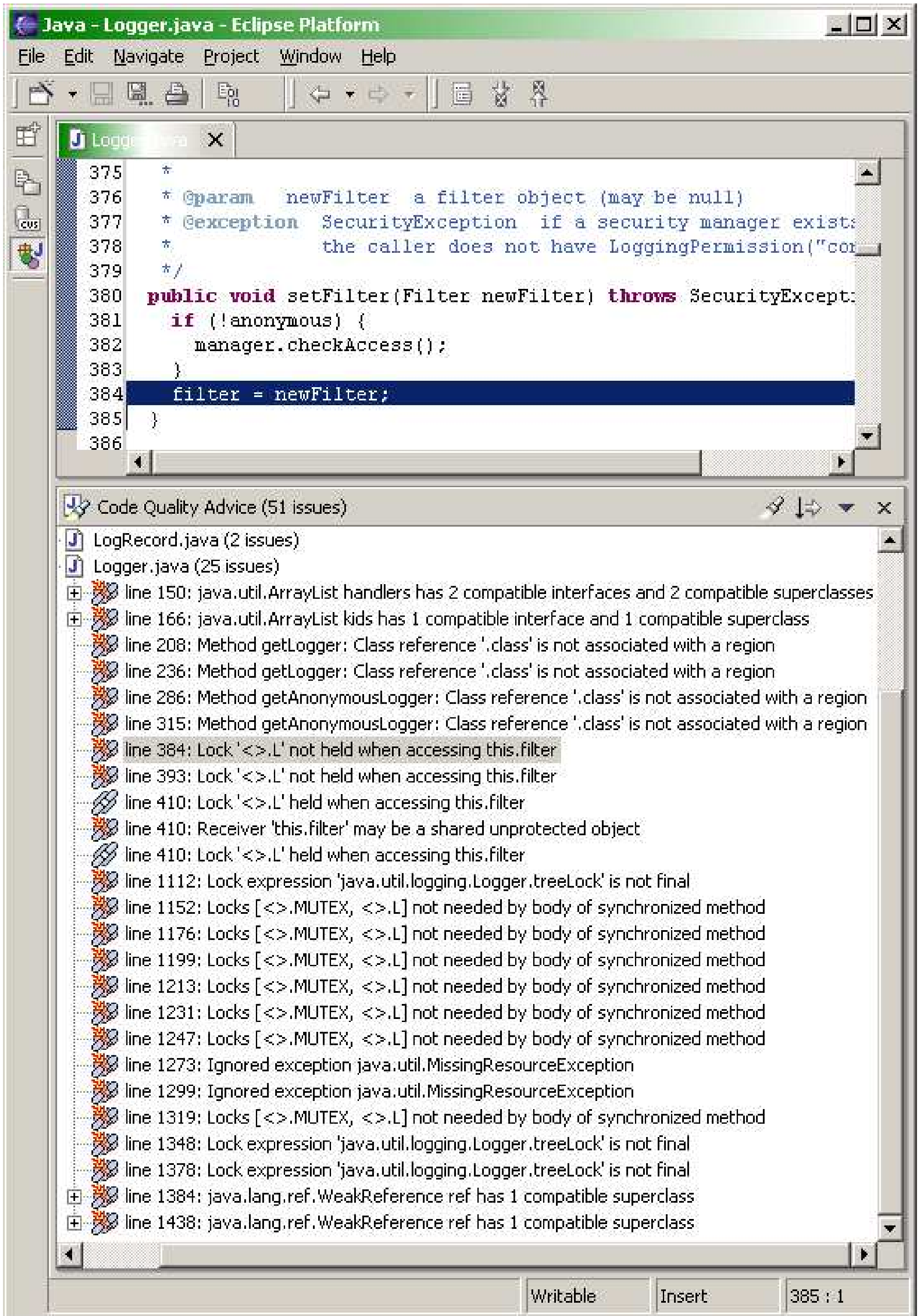


Figure 7.12: Analysis results after protecting field `filter`.

the condition-variable–related methods). Thus, the synchronization is suspicious, and suggests that we may need to associate the lock `this` with additional state.

- There is a single warning that the object referenced by `filter` may be a shared unprotected object. This comes from the method invocation shown on line 408 in Figure 7.11.

It turns out that there is a real race condition in the implementation of `Logger`, which we have identified as a consequence of this study.[5] Consider method `setFilter` as shown in Figure 7.12 and method `log` as shown in Figure 7.11. Method `setFilter` does not acquire the lock before assigning a new value that is *specifically allowed to be `null`* to `filter`. This can cause a null-pointer exception in `log` because clearly the intent of the `synchronized` block that it contains is to prevent `filter` from becoming `null` once it has been verified to be non-`null`. To be compliant with the locking model and thus to avoid this possibility, methods `setFilter` and `getFilter`, the methods containing the unprotected uses of `filter` identified above, are declared to be `synchronized`.

After introducing the missing synchronization, analysis again reports 19 concurrency-related issues, except that now the two assurance failures regarding `setFilter` and `getFilter`, on lines 384 and 393 respectively, are replaced by positive assurances; see Figure 7.13. The race condition over field `filter` is now fixed. We do not address the seven *warnings* related to `filter` and lock `L` until later in this example.

7.6.2 A Global Tree Lock

We continue by documenting the design intent behind additional existing critical sections. Still referring to Figure 7.13, we see that analysis warns that on lines 1112, 1348, and 1378 "Lock expression 'java.util.logging.Logger.treeLock' is not final." From these three warnings we learn that

- The object referenced by the `static` field `treeLock` is used as a lock. *The state protected by this lock is not yet unidentified.*
- The field `treeLock` is not `final`, making it *unsafe to use as a lock* because the object that it refers to can change over time.

An obvious first step for us to take to address these warnings is to declare the field `treeLock` to be `final`. While doing this, we find that the declaration of the field is informally annotated with design intent indicating that the fields `parent`, `kids`, and `levelObject` of all `Logger` instances

[5] We reported this race condition to the java bug database: `http://developer.java.sun.com/developer/bugParade/`. It has been assigned the bug id 4779253.

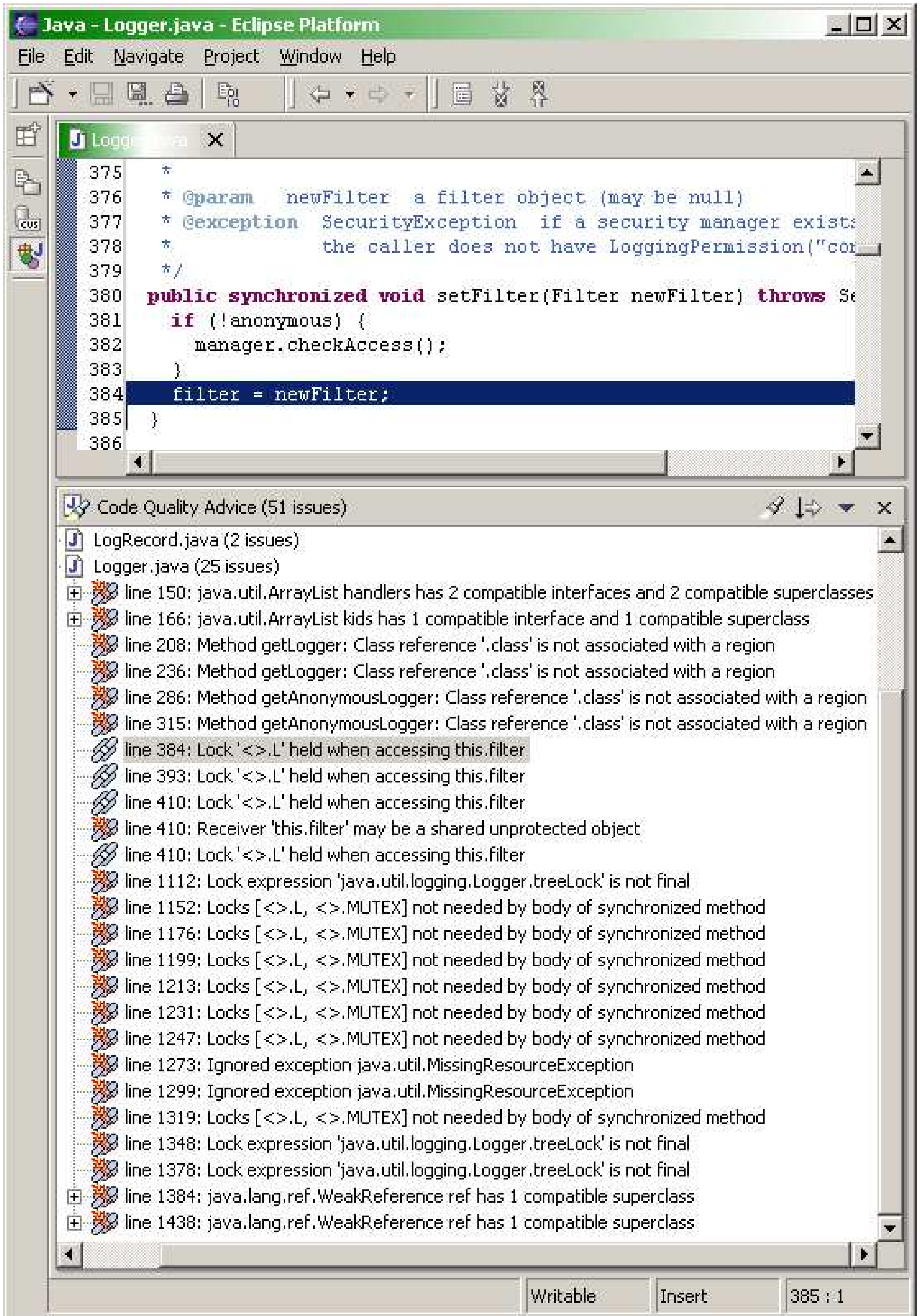


Figure 7.13: Analysis results after fixing the race condition over `filter`.

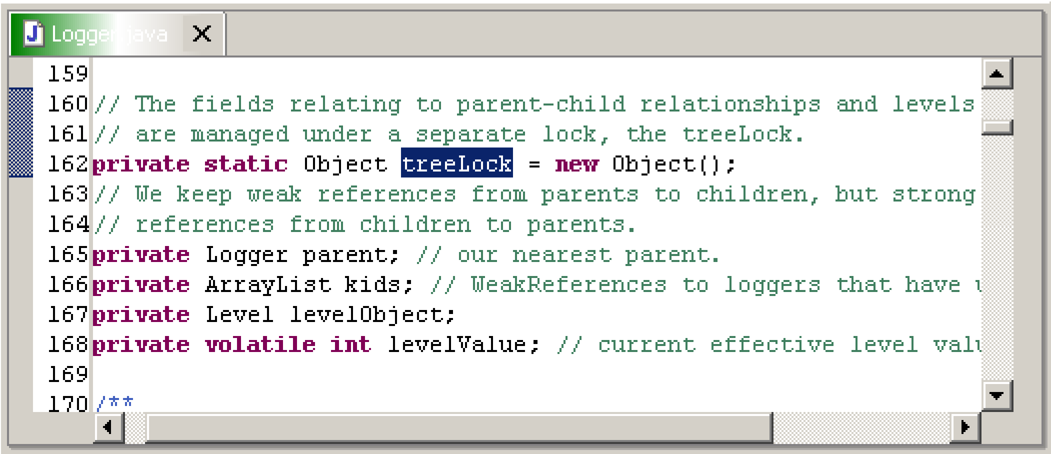


Figure 7.14: Design intent surrounding field `treeLock`.

are intended to be protected by the lock on the object referenced by `Logger.treeLock`; see Figure 7.14.

We thus take the following actions on the source code:

- We declare the `static` field `treeLock` to be `final`.
- We declare a new `static` region abstracting the logger hierarchy information:

  ```
  @region public static LoggerHierarchy
  ```

- We populate the new region with the fields `parent`, `kids`, and `levelObject`. We also aggregate the state of the list referenced by `kids` into the `LoggerHierarchy` region. We *cannot* do the same for the `parent` field because `Logger` objects *are possibly aliased*, if for no other reason than by virtue of having multiple children.
- We associate the lock represented by `treeLock` with the new region:

  ```
  @lock TreeLock is treeLock protects LoggerHierarchy
  ```

Reanalyzing the program with the additional design intent results in 37 concurrency-related issues (out of a total of 43 issues) in class `Logger`. The three earlier warnings regarding `treeLock` have been replaced with 21 new analysis results, highlighted in Figure 7.15:

- Twelve positive assurances that fields `levelObject`, `parent`, and `kids` are accessed according to the annotated locking model.
- Eight potential violations of the locking model where `TreeLock` is needed but not known to be held.
- An additional warning on line 1426 that `levelObject` "may be a shared unprotected object."

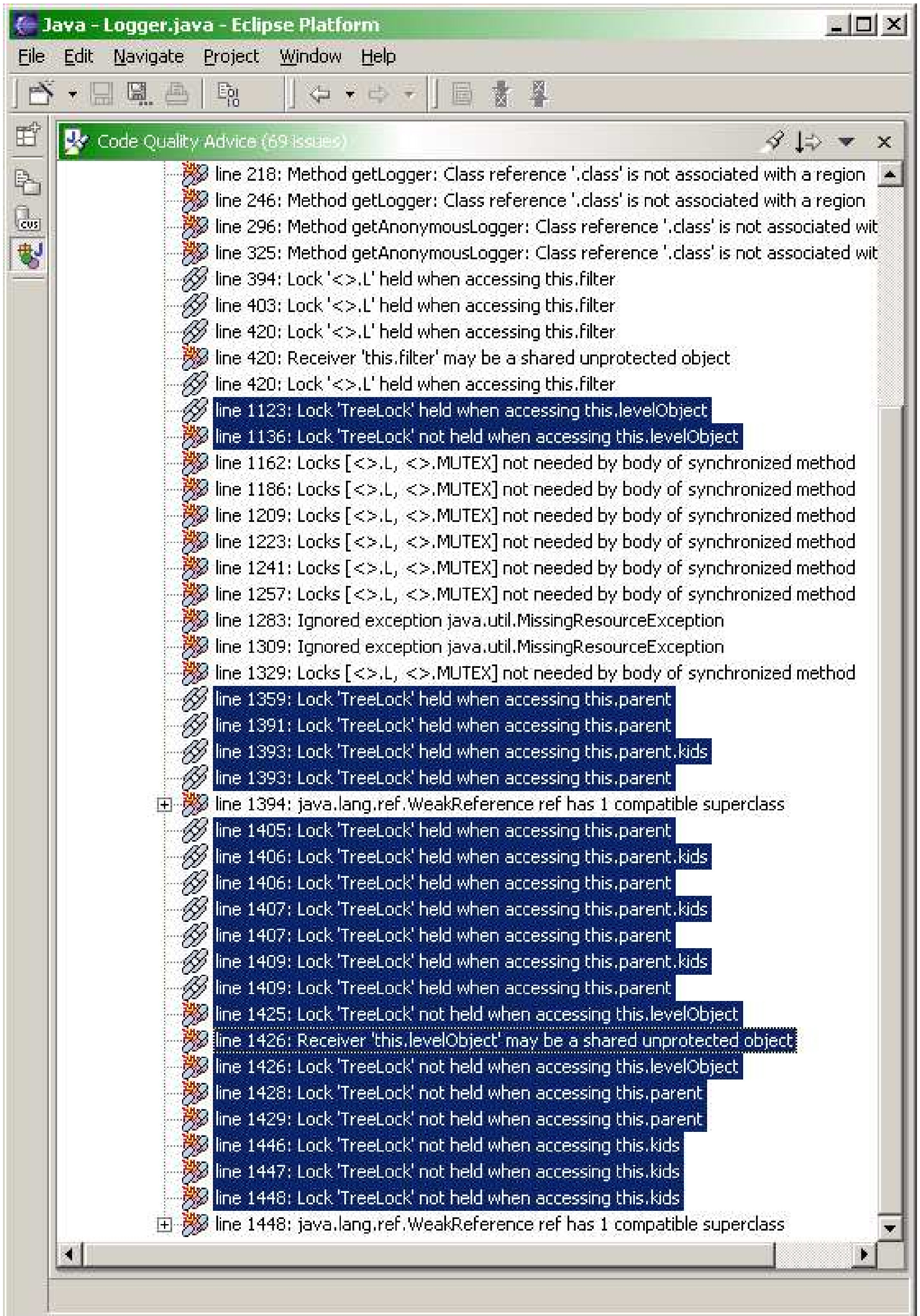


Figure 7.15: Analysis results after declaring the shared region `LoggerHierarchy`.

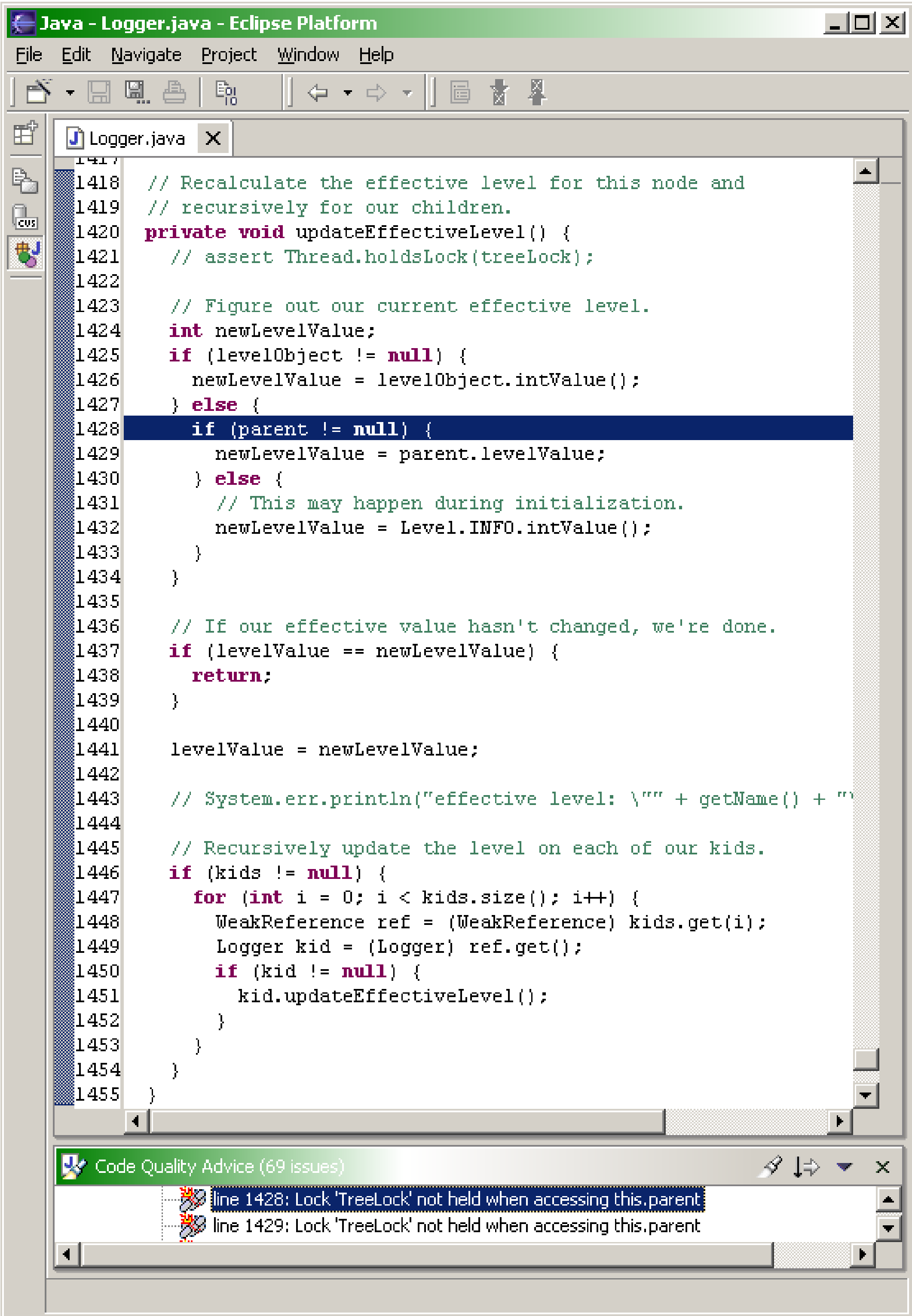


Figure 7.16: Method `updateEffectiveLevel`. Note the assertion on line 1421.

Investigation of the eight locking model violations reveals that `getLevel`, the getter method for field `levelObject`, does not acquire `TreeLock`. We modify the method to conform to the model:

```
public Level getLevel() {
  synchronized(treeLock) { return levelObject; }
}
```

This takes care of the model violation associated with line 1136. The other seven potential locking model violations all occur within the implementation of method `updateEffectiveLevel`, a `private` helper method. We are lucky: the method is already *informally* annotated with design intent, see line 1421 in Figure 7.16. A comment contains the assertion that the current thread should hold the lock on the object referenced by `treeLock`. We declare this design intent *formally* by annotating the method with

```
@requiresLock TreeLock
```

Reanalysis now assures that the callers of the method adhere to the newly extended locking model. The complete results are omitted; of particular interest here are (1) the three positive assurances—and no assurance failures—that `updateEffectiveLevel` is called consistently with the locking model; and (2) the 20 positive assurances that the fields in region `LoggerHierarchy` are accessed consistently with the locking model.

Regarding the protection of state in region `LoggerHierarchy`, it remains to address the warning that field `levelObject` may be a shared unprotected object. This warning originates from the method call "`levelObject.intValue()`" on line 1426 in Figure 7.16. Here, analysis requires more information about the thread-safety of `Level` objects. A brief inspection of `Level` (source code omitted) suggests that the class is intended to be a type-safe enumeration of immutable objects [Blo01a]. Thus, `Level` objects are sharable by multiple threads because they are *intended* to be immutable. We annotate class `Level` with this intent by adding the annotation

```
@selfProtected
```

(Recall that this annotation declares that the implementation of the class is completely responsible for insuring its own thread-safety.) Analysis now reports 45 issues in `Logger`, 39 of which are concurrency-related. Again, we omit the complete results, but of specific interest is that the warning about the possibly shared use of `levelObject` is no longer reported.

7.6.3 The Locking Model Thus Far

We have now (1) documented a subset of the locking model for `Logger` objects, and (2) assured that the source code is consistent with the model. We used an iterative process of introducing small numbers of annotations, fixing code to be consistent with the expressed design intent, and then repeating analysis incrementally to and assure source code–model consistency. Specifically we have declared the following model, shown graphically in Figure 7.22(a).

- The region `filter` is protected by locking the `Logger` object itself. This lock is named "`L`."
- The region `LoggerHierarchy`, the parent region of fields `parent`, `kids`, and `level-Object` of *all* `Logger` objects, is protected by locking the object referenced by the `static` field `Logger.treeLock`. This lock is named "`TreeLock`."
- The state of the list referenced by the `kids` field is aggregated into the region `Logger-Hierarchy`, and thus also protected by `TreeLock`.
- The caller of the method `updateEffectiveLevel` must acquire `TreeLock`.

7.6.4 A Policy Lock

We now turn our attention to four warnings, originally reported on lines 206, 234, 284, and 313 in Figure 7.11, that we have ignored thus far: "Class reference '.class' is not associated with a region." These warnings identify four `static synchronized` methods: methods that acquire the lock on the object referenced by `Logger.class`. This object has not been explicitly declared to represent a lock, and thus the analysis is unable to assure anything about the critical sections defined by these methods. The four methods are factory methods for getting references to `Logger` objects, and have the same general form; Figure 7.17 shows, for example, `getLogger(String).` These are the only places where the lock represented by the object referenced by`Logger.class` is acquired; there would be additional warnings were it used elsewhere. There are no fields that should obviously be associated with the lock; in particular, no mutable `Logger` state is directly accessed in the `getLogger` method in Figure 7.17. Given this information and the similar structure of all the methods, we infer the design intent that the lock is used to prevent a race condition during the creation `Logger` objects that could cause multiple `Logger` instances with the same name to be created, a violation of higher-level abstractions in the API of the package.

This use of the object referenced by `Logger.class` is consistent with our notion of policy locks: locks that are not associated with state, but that are used to enforce consistency with a class's internal concurrency policy. To document that this is the intent behind this use of `Logger.class`,

```
public static synchronized Logger getLogger(String name) {
  LogManager manager = LogManager.getLogManager();
  Logger result = manager.getLogger(name);
  if (result == null) {
    result = new Logger(name, null);
    manager.addLogger(result);
    result = manager.getLogger(name);
  }
  return result;
}
```

Figure 7.17: The factory method `Logger.getLogger(String)`.

and as a result to suppress the four warnings, we add to class `Logger` the annotation

```
@policyLock SerializeLoggerCreation is class
```

Analysis now reports 35 concurrency-related issues, highlighted in Figure 7.18:

- Three positive assurances that the lock precondition on `updateEffectiveLevel` is respected.
- Twenty-four positive assurances that fields are accessed according to the locking model.
- Seven warnings that lock `L` is acquired but not needed. Specifically, the methods `addHandler`, `removeHandler`, `getHandlers`, `setUseParentHandlers`, `getUseParentHandlers`, `findResourceBundle`, and `setupResourceInfo` are `synchronized` but do not use the field `filter`, the sole field associated with the lock represented by `this`.
- One warning that `filter` may be a shared unprotected object.

7.6.5 Expanding the Locking Model

The seven warnings about `synchronized` methods suggest that our reverse-engineered locking model needs to be expanded to include more state. Five of the methods involved in the warnings, shown in Figure 7.19, are related to managing a list of `Handler` objects associated with the `Logger`. A brief inspection of the methods, and some surrounding code, suggests additional design intent:

- Field `manager` is immutable, and ought to be declared `final`.
- Field `handlers` is protected by the lock represented by `this`, as is the state of the `ArrayList` that it references.
- Field `useParentHandlers` is protected by the lock represented by `this`.

We document this intent, which replaces our previous locking model, by

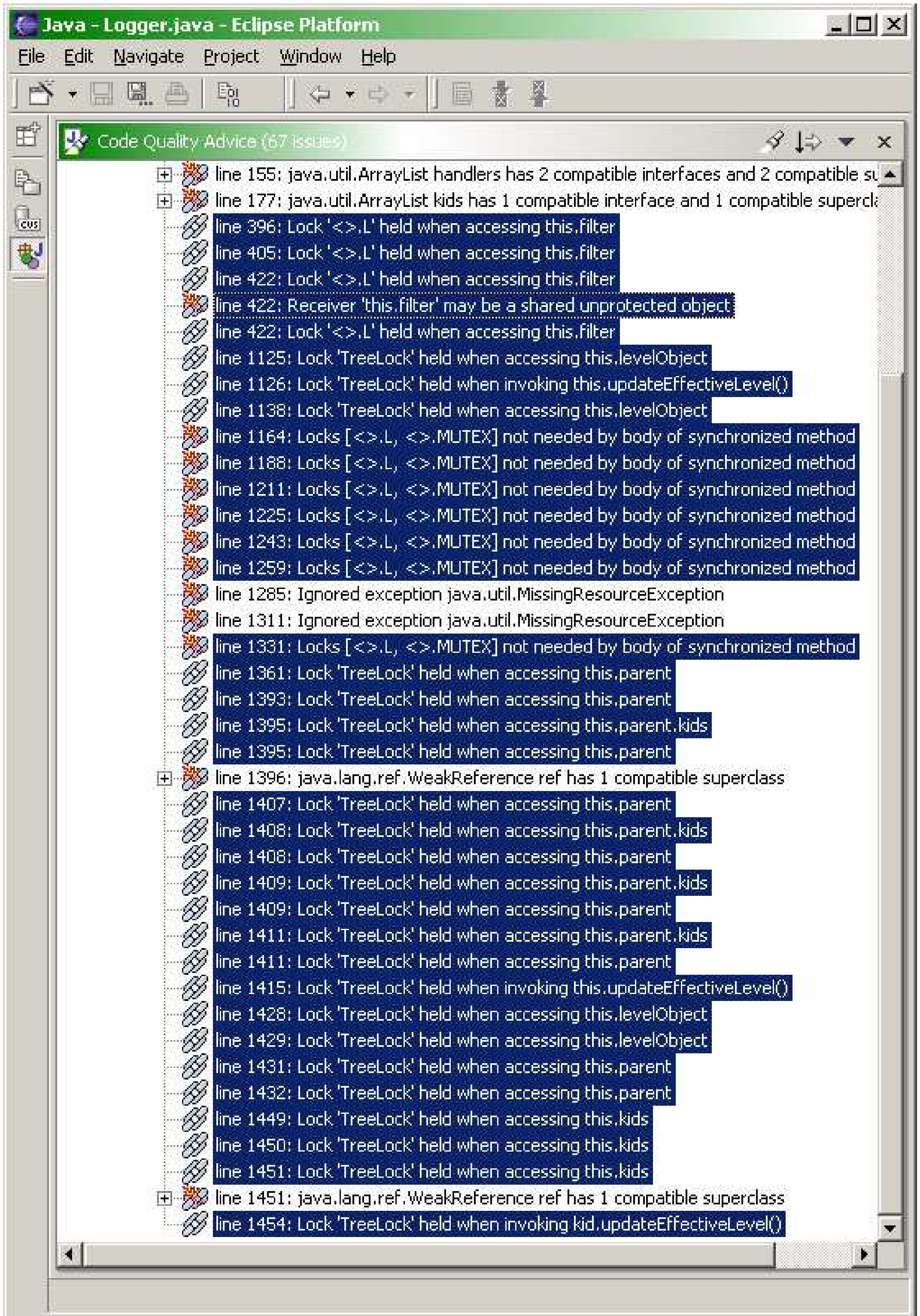


Figure 7.18: Analysis results after declaring `Logger.class` to represent a policy lock.

```
public synchronized void addHandler(Handler handler)
throws SecurityException {
  // Check for null handler
  handler.getClass();
  if (!anonymous) {
    manager.checkAccess();
  }
  if (handlers == null) {
    handlers = new ArrayList();
  }
  handlers.add(handler);
}

public synchronized void removeHandler(Handler handler)
throws SecurityException {
  if (!anonymous) {
    manager.checkAccess();
  }
  if (handler == null) {
    throw new NullPointerException();
  }
  if (handlers == null) {
    return;
  }
  handlers.remove(handler);
}

public synchronized Handler[] getHandlers() {
  if (handlers == null) {
    return emptyHandlers;
  }
  Handler result[] = new Handler[handlers.size()];
  result = (Handler[]) handlers.toArray(result);
  return result;
}

public synchronized void
setUseParentHandlers(boolean useParentHandlers) {
  if (!anonymous) {
    manager.checkAccess();
  }
  this.useParentHandlers = useParentHandlers;
}

public synchronized boolean getUseParentHandlers() {
  return useParentHandlers;
}
```

Figure 7.19: The five “handler”-related methods from class `Logger`.

```
/**
 * A Filter can be used to provide fine grain control over
 * what is logged, beyond the control provided by log levels.
 * <p>
 * Each Logger and each Handler can have a filter associated with it.
 * The Logger or Handler will call the isLoggable method to check
 * if a given LogRecord should be published.  If isLoggable returns
 * false, the LogRecord will be discarded.
 *
 * @version 1.3, 12/03/01
 * @since 1.4
 */
public interface Filter {
  /**
   * Check if a given log record should be published.
   * @param record  a LogRecord
   * @return true if the log record should be published.
   */
  public boolean isLoggable(LogRecord record);
}
```

Figure 7.20: The `Filter` interface.

- Declaring that `manager` is `final`.
- Declaring a new region, via annotation, `LoggerInfo`, and populating it with the fields `filter`, `handlers`, and `useParentHandlers`.
- Annotating that `handlers` is unaliased, and that the state of the list should be aggregated into the state of the `Logger`:

```
/**
 * @mapInto LoggerInfo
 * @unshared
 * @aggregate Instance into LoggerInfo
 */
private ArrayList handlers;
```

- Replacing our original `@lock` annotation

```
@lock L is this protects filter
```

with

```
@lock InfoLock is this protects LoggerInfo
```

A graphical representation of the modified model is shown in Figure 7.22(b).

We also now deal with the long-ignored warning that `filter` refers to a possibly shared unprotected object. The field is of type `Filter`, an interface shown in Figure 7.20. Clearly implementations of the class must be thread-safe: after all, the filter may be used by many `Logger` objects. We thus declare the design intent that implementations of `Filter` should protect themselves by annotating the interface with `@selfProtected`.

```
// Private utility method to map a resource bundle name to an
// actual resource bundle, using a simple one-entry cache.
// Returns null for a null name.
// May also return null if we can't find the resource bundle and
// there is no suitable previous cached value.
private synchronized ResourceBundle findResourceBundle(String name) {
  // Return a null bundle for a null name.
  if (name == null) { return null; }

  Locale currentLocale = Locale.getDefault();

  // Normally we should hit on our simple one entry cache.
  if (catalog != null
    && currentLocale == catalogLocale
    && name == catalogName) {
    return catalog;
  }

  // Use the thread's context ClassLoader.  If there isn't one,
  // use the SystemClassloader.
  ClassLoader cl = Thread.currentThread().getContextClassLoader();
  if (cl == null) { cl = ClassLoader.getSystemClassLoader(); }
  try {
    catalog = ResourceBundle.getBundle(name, currentLocale, cl);
    catalogName = name;
    catalogLocale = currentLocale;
    return catalog;
  } catch (MissingResourceException ex) {
    // Woops.  We can't find the ResourceBundle in the default
    // ClassLoader.  Drop through.
  }

  // Fall back to searching up the call stack and trying each
  // calling ClassLoader.
  for (int ix = 0;; ix++) {
    Class clz = sun.reflect.Reflection.getCallerClass(ix);
    if (clz == null) {
      break;
    }
    ClassLoader cl2 = clz.getClassLoader();
    if (cl2 == null) { cl2 = ClassLoader.getSystemClassLoader(); }
    if (cl == cl2) {
      // We've already checked this classloader.
      continue;
    }
    cl = cl2;
    try {
      catalog = ResourceBundle.getBundle(name, currentLocale, cl);
      catalogName = name;
      catalogLocale = currentLocale;
      return catalog;
    } catch (MissingResourceException ex) {
      // Ok, this one didn't work either.
      // Drop through, and try the next one.
    }
  }

  if (name.equals(catalogName)) {
    // Return the previous cached value for that name.
    // This may be null.
    return catalog;
  }
  // Sorry, we're out of luck.
  return null;
}
```

Figure 7.21: The method `findResourceBundle`. Uses of fields `catalog`, `catalogName`, and `catalogLocale` are in boldface.

The remaining two methods that acquire `InfoLock` but that do not access any state in its associated region are `findResourceBundle` and `setupResourceInfo`. Method `findResourceBundle`, whose implementation is shown in Figure 7.21, can be executed in multiple threads at once. It is invoked, for example, by several of the public logging methods. It is clear that the values of `catalog`, `catalogName`, and `catalogLocale` should be kept in sync with each other, and thus make up the state being protected during execution of the method. We add this design intent to the class by

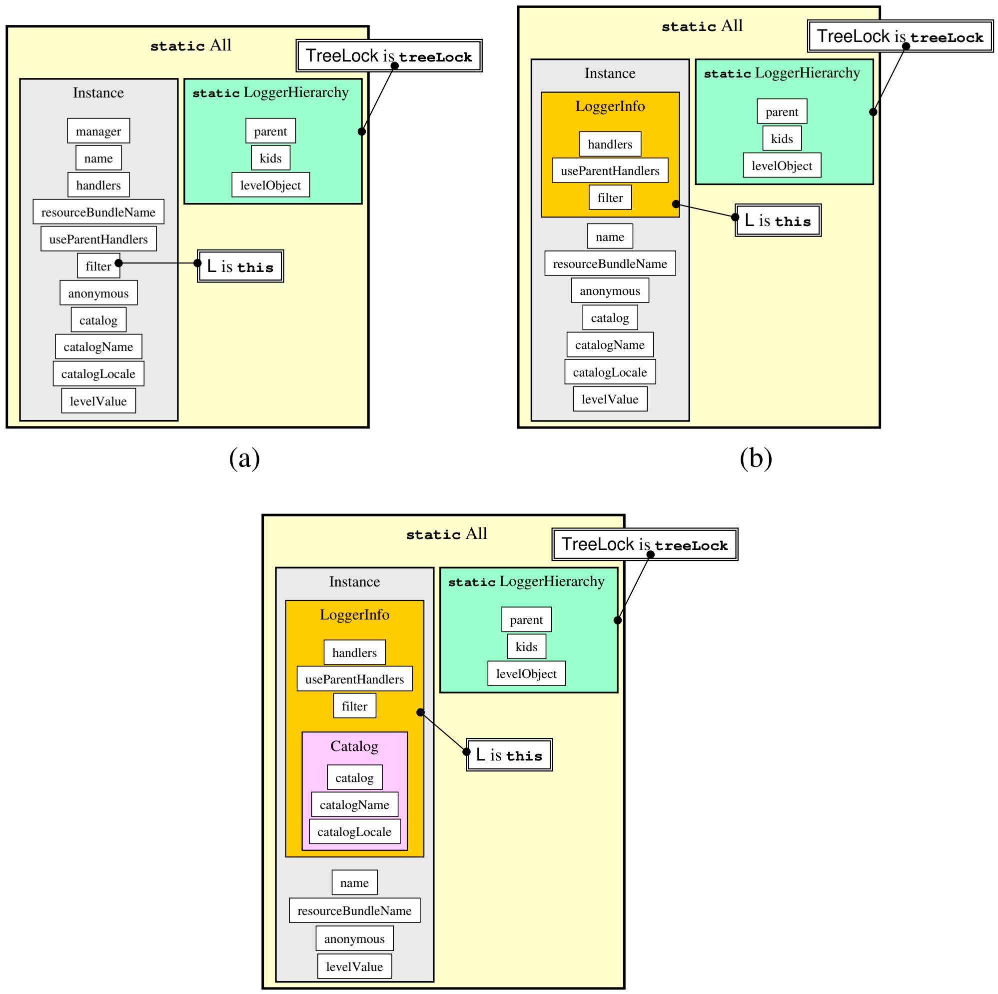
- Declaring a new region `Catalog` as a subregion of `LoggerInfo`.
- Populating the new region with the fields `catalog`, `catalogName`, and `catalogLocale`.

The modified model is shown graphically in Figure 7.22(c).

7.6.6 Unclear Intent

Method `setupResourceInfo`, shown in Figure 7.23, is more interesting despite being a very simple method that only (1) calls `findResourceBundle` to check that the named resource exists, and (2) assigns the name of the resource to the field `resourceBundleName`. The obvious state being protected is the aforementioned field. The associated getter method, however, is not declared to be `synchronized`, and this should be recognized as an error. Were we to add `resourceBundleName` to our protected region `LoggerInfo`, the getter would be identified as being inconsistent with the model. However, the only other direct uses of the field are in one of the `getLogger` factory methods, which is also the only place outside of the constructor where `setupResourceInfo` is called. As we have seen already, `getLogger` is already `synchronized`, and thus any additional locking to protect `resourceBundleName` is overkill. But we cannot have both `this` and `Logger.class` protect the field.

What is the intent of the designers of the class is not clear at this point. We proceed with our assurance process based on the following additional observation: the value of the field `resourceBundleName` *does not* need to be kept in sync with the values of any other fields. It only needs synchronization (1) for higher-level policy decisions, but this already correctly occurs in `getLogger`, and (2) for ensuring that changes to its value propagate in Java's memory model. But this later task can *also* be accomplished by declaring the field to be `volatile`. Finally, there do not appear to be any consistency constraints—*i.e.*, internal concurrency policy concerns—that require the invocation of `findResourceBundle` and the update of `resourceBundleName` to occur within a single critical section. We conclude that the best way to address the remaining warning regard-



(c)

Figure 7.22: Evolution of the state and locking model for `Logger`. These diagrams do not account for policy locks or state aggregation. (a) The model after protecting `filter` and identifying region `LoggerHierarchy`. (b) The model after expanding the protected state to include the newly defined `LoggerInfo` region. The field `manager` is removed from the model because it has been declared to be `final`. (c) The model after identifying the `Catalog` region.

```
private synchronized void setupResourceInfo(String name) {
  if (name == null) {
    return;
  }
  ResourceBundle rb = findResourceBundle(name);
  if (rb == null) {
    // We've failed to find an expected ResourceBundle.
    throw new MissingResourceException(
      "Can't find " + name + " bundle", name, "");
  }
  resourceBundleName = name;
}
```

Figure 7.23: The method `setupResourceInfo`.

ing `setupResourceInfo` is by (1) declaring the field `resourceBundleName` to be `volatile`, which removes it from the scope of the locking model; and (2) removing the `synchronized` declaration from `setupResourceInfo`.

Analysis now gives only positive assurance results regarding concurrency related properties of `Logger`; see Figure 7.24. We have thus used our tool and iterative assurance process to

- Document the locking model used by class `Logger`.
- Assure consistency between this model and the source code.
- Discover a race condition that could result in an exception.
- Discover several "minor" race conditions that can result in inconsistent views of memory.
- Uncover an piece of policy intent that probably needs further consideration by the designers of the class.

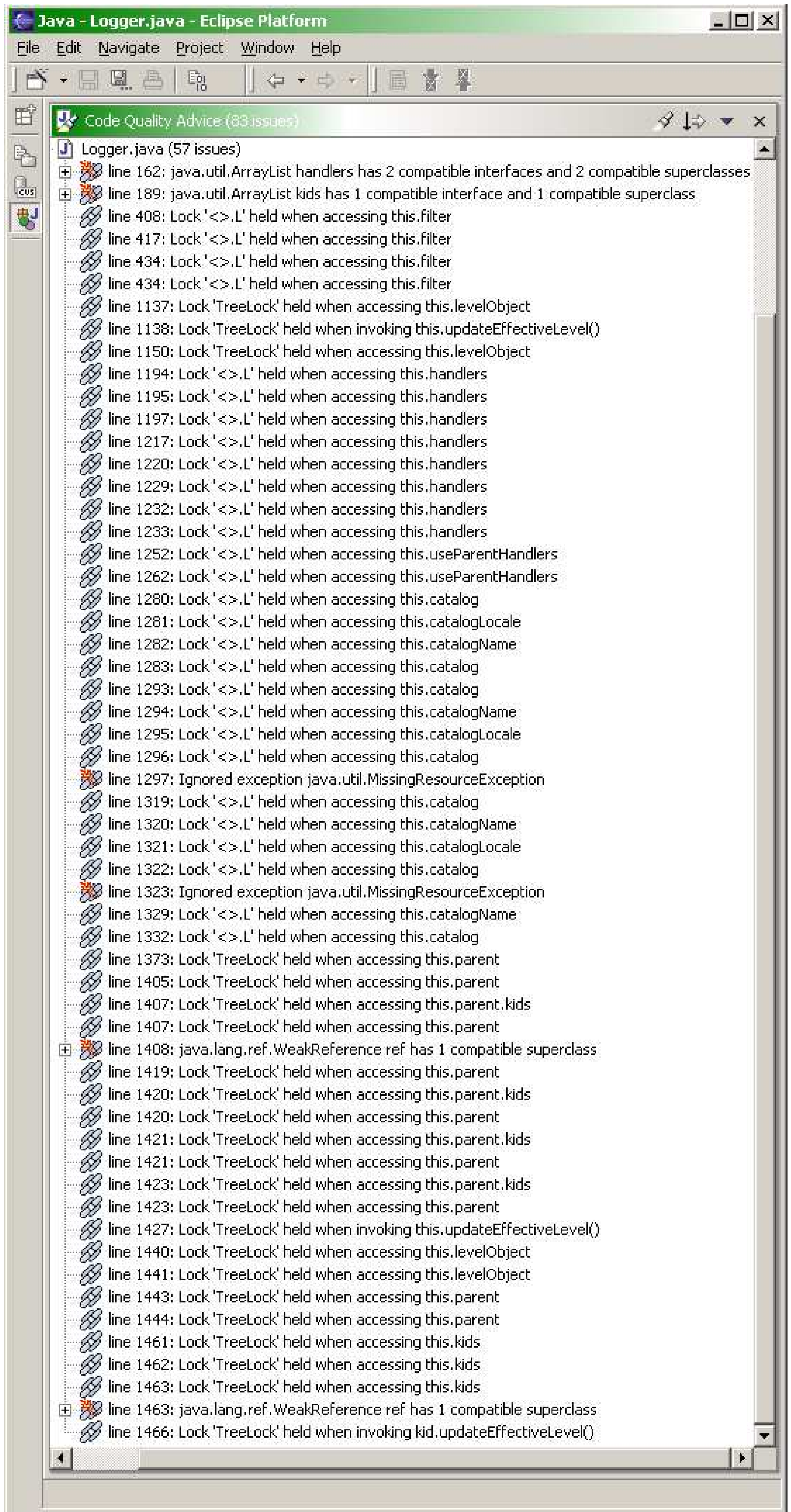


Figure 7.24: Analysis results assuring `Logger` to be consistent with its declared locking model.

Chapter 8

Towards a Generative Approach to Concurrency Management

The assurance techniques described in the previous chapters provide the foundations for using program transformation techniques to manage the concurrent attributes of a program. This chapter is a speculative excursion, and elaborates some of the ideas of this *generative approach to concurrency management* using examples of notional transformation scenarios and describes preliminary results on the required transformations. Semantics-based, behavior-preserving, source-level, concurrency-related program transformations are a fundamental building block of the generative approach. Our usage scenario is that, in a programming environment, the application of a transformation is initiated by the programmer, but it is performed by a software tool along with the necessary analyses, relying on program annotations expressing programmer design intent. A transformation, by *our* definition, is meaning-preserving with respect to the programmer-specified concurrency-related design intent. This means that a transformation does not result in inappropriate accesses to shared regions, and that concurrency policy is respected. In general, deadlock avoidance is also an issue. This dissertation does not present techniques for capturing design intent regarding deadlock avoidance, and thus we do not consider deadlock any further in this chapter.

We first review what is the generative approach, and give a sample of basic transformations that make up the approach. We then present a series of examples that demonstrate the use of transformations within the generative approach. Our intent is to expose the difficulty in determining the proper preconditions for soundness of a transformation—what we describe are not always transformations by the above definition because they may fail to preserve program safety relative to the programmer's intent. Through these examples, we also describe some of the techniques a tool might use to insure safety, and identify the division of labor between the programmer and tool. The examples suggest the operation of many of the transformations. We follow the examples with a brief elaboration of the operation of and issues raised by two transformations, **split lock** and **shrink critical section**. Finally, we discuss related work on transformational approaches to concurrent programming.

8.1 The Generative Approach

The goal of the generative approach to concurrency management is to establish a principled approach to the introduction and management of concurrency, allowing the trading off of performance and concurrency to be explored without disturbing functionality and while keeping program complexity manageable. Concurrency-related source-level program transformations by themselves are not enough to maintain the safety of a concurrent program as it evolves in a practical engineering approach. The transformations themselves must be applied with respect to a programming discipline that further insures that the annotations used by the transformations are correct and stay correct, and that manages the introduction and evolution of concurrency policy. The generative approach thus consists of applying program transformations in a framework of assurance and annotation management. Adherence to the generative approach insures that a concurrent program is always free of race conditions as defined by a programmer specified policy. It is based on the observation that it is easier to stay in such a state than to initially arrive at—be proven to be in—such a state given an arbitrary concurrent program. So that the concurrent program is known to be initially safe, the generative approach prescribes a specific technique for increasing the extent of concurrency in programs: they must be generated from initial sequential programs.

Transformations are performed by the tool on the behalf of the programmer and their safety is verified using program analyses supported by programmer-specified design intent. It is our belief that tool support is essential to making our approach practicable. Responsibility for design intent and safety is partitioned between the programmer and the tool: *the tool strictly regulates* the introduction and management of concurrency-related annotations and transformations, while *the programmer is in complete control* of the original partitioning of a class into regions and other expressions of design intent.

We simplify the following discussion by only considering the transformation of `final` classes, that is, classes that cannot have subclasses.

8.1.1 Catalog of Transformations

We have identified a number of transformations that are fundamental to the generative approach to concurrency management. The generative approach is first applied to a non-concurrent implementation of a class using the **synchronize class** transformation. The class is reimplemented as a simple monitor and given a default concurrency policy with which it is already compliant. The extent of concurrency supported by the class can then be manipulated by a variety of transformations. **Split lock** decreases the granularity at which state is protected by moving it "down" the region hierarchy. A single shared region is replaced with many shared regions based on its subregions. The granularity of protection may be moved "up" the region hierarchy using **merge locks**, which replaces multiple shared regions with a single ancestor shared region. The scope of a critical section may be altered using **shrink critical section**.

Additional opportunities for allowing method interleaving may be introduced using **split critical section** to convert a single `synchronized` block into a sequence of `synchronized` blocks. The dual **merge critical sections** may be used to remove interleaving opportunities. Altering the programmer-specified concurrency policy based on the modified interleaving opportunities is a separate step in the generative process. The **implement policy** transformation is used to reify the policy in the implementation.

Locking responsibility may be modified using the **synchronize method** and **synchronize callsite** transformations, which move the responsibility to the implementation or the caller, respectively. These transformations affect a wider body of code because of the necessity of identifying and updating method callsites.

Our approach to annotation, analysis, and transformation is based on regions as shared resources. Transformations that affect the region hierarchy and the aggregation of state are also basic building blocks of the generative approach.

8.2 Evolving `EventQueue`

We now use the generative approach to increase the amount of concurrency exploitable by clients of the class `EventQueue`. This class implements a simple queue that buffers events, represented as `Objects`. Listeners register themselves with the queue using implementations of the `EQListener` interface, which receives events from the queue using the `dequeued` callback. Figure 8.1 shows the class after it has been made initially concurrent using the **synchronize class** transformation.

This transformation modifies the original sequential version of the class by (1) declaring a new lock that protects the `Instance` region, see line 3; (2) declaring each non-`private` method to be `synchronized`; and (3) declaring a lock precondition on each `private` method, such as on line 46.

Instances of `EventQueue` have three abstract instance regions: (1) `Instance`, inherited from `Object`; (2) `Listeners`, a subregion of `Instance`, contains the state of the `List` referenced by `listeners`; and (3) `Queue`, a subregion of `Instance`, contains the field `len` and the state of the `List` referenced by `queue`. To increase the amount of concurrent access supported by the class we wish to (1) modify the granularity of locking based on the region hierarchy so that the queue's listeners and the queue's data are separately protected, and (2) minimize the scope of the resulting critical sections.

8.2.1 Splitting the Lock

Splitting a lock alters a program so that the state within a shared region is protected at the granularity of the subregions of the original region. For our example, this means replacing the protection of the single region `Instance` by the protection of the two subregions `Queue` and `Listeners`. The transformation associated new lock objects with subregions and introduces the appropriate annotations to reflect the modified design intent. A precondition of the transformation is that all subregions are identified with a `final` field of the class or the instance's receiver to provide the lock object to be used to protect the subregion. For our example, the object referred to by the field `listeners` will protect region `Listeners`, and the object referred to by the field `queue` will protect region `Queue`.

Generally, the code is changed as follows. (1) All the critical sections for the lock being split are identified via use of the `@lock` annotations. (2) Effects analysis is used to identify the most specific regions affected by the critical section. (3) The subregions of the original region affected by the critical section determine how the critical section is modified to use the new region–lock assignments. For example, the body of method `addEQListener` has the effect `reads Listeners`, so it is modified to lock on `listeners` instead of `this`. In a similar manner, the `@requiresLock` annotations are updated. The modified `EventQueue` class is shown in Figure 8.2.

```
/** @region public Listeners
 *  @region public Queue
 *  @lock Lock is this protects Instance  */
public final class EventQueue {
  /** @unshared
   *  @aggregate Instance into Listeners */
  private final List listeners;
  /** @unshared
   *  @aggregate Instance into Queue */
  private final List queue;
  /** @mapInto Queue */
  private int len;

  public EventQueue() {
    listeners = new ArrayList();
    queue = new LinkedList();
    len = 0;
  }

  /** @writes Listeners */
  public synchronized void addEQListener( EQListener l ) { listeners.add( l ); }

  /** @writes Listeners */
  public synchronized void removeEQListener( EQListener l ) { listeners.remove( l ); }

  /** @writes All
   *  @requiresLock Lock */
  private void fireEQEvent( Object o ) {
    List copy = (List)((ArrayList)listeners).clone();
    for( int i = 0; i < copy.size(); i++ ) {
      final EQListener l = (EQListener)copy.get( i );
      l.dequeued( o );
    }
  }

  /** @reads Queue */
  public synchronized int getSize() { return len; }

  /** @writes Queue */
  public synchronized void enqueue( Object o ) {
    queue.add( o );
    len += 1;
  }

  /** @writes Queue
   *  @requiresLock Lock */
  private Object dequeue() {
    if( len == 0 ) return null;
    else {
      len -= 1;
      return queue.remove( 0 );
    }
  }

  /** @writes All */
  public synchronized void dispatchEvent() {
    final Object o = dequeue();
    fireEQEvent( o );
  }
}
```

Figure 8.1: Initial version of `EventQueue` that uses a single lock to protect the entire object.

```
/** @region public Listeners
 *  @region public Queue
 *  @lock QLock is queue protects Queue
 *  @lock ListLock is listeners protects Listeners */
public final class EventQueue {
  /** @unshared
   *  @aggregate Instance into Listeners */
  private final List listeners;
  /** @unshared
   *  @aggregate Instance into Queue */
  private final List queue;
  /** @mapInto Queue */
  private int len;

  public EventQueue() {
    listeners = new ArrayList();
    queue = new LinkedList();
    len = 0;
  }

  /** @writes Listeners */
  public void addEQListener( EQListener l ) { synchronized( listeners ) { listeners.add( l ); } }

  /** @writes Listeners */
  public void removeEQListener( EQListener l ) { synchronized( listeners ) { listeners.remove( l ); } }

  /** @writes All
   *  @requiresLock ListLock */
  private void fireEQEvent( Object o ) {
    List copy = (List)((ArrayList)listeners).clone();
    for( int i = 0; i < copy.size(); i++ ) {
      final EQListener l = (EQListener)copy.get( i );
      l.dequeued( o );
    }
  }

  /** @reads Queue */
  public int getSize() { synchronized( queue ) { return len; } }

  /** @writes Queue */
  public void enqueue( Object o ) {
    synchronized( queue ) {
      queue.add( o );
      len += 1;
    }
  }

  /** @writes Queue
   *  @requiresLock QLock */
  private Object dequeue() {
    if( len == 0 ) return null;
    else {
      len -= 1;
      return queue.remove( 0 );
    }
  }

  /** @writes All */
  public void dispatchEvent() {
    synchronized( queue ) {
      synchronized( listeners ) {
        final Object o = dequeue();
        fireEQEvent( o );
      }
    }
  }
}
```

Figure 8.2: `EventQueue` after the split-lock modification. Differences from Figure 8.1 are underlined.

8.2.2 Shrink Critical Section

Another concurrency-related modification is to shrink the computational scope of a critical section. In general, applying this modification to a set of locks would identify the `synchronized` statements that use those locks, and for each one, alter the lexical positions of the beginning and ending of the critical sections in the program text to remove any statements that do not affect the region associated with the critical section's lock. For example, applying it to class `EventQueue` in Figure 8.2 for `QLock` and `ListLock` would only affect one method, `dispatchEvent`, transforming it to be

```
1  /** @writes All */
2  public void dispatchEvent() {
3    Object temp;
4    synchronized( queue ) { temp = dequeue(); }
5    final Object o = temp;
6    synchronized( listeners ) { fireEQEvent( o ); }
7  }
```

because method `dequeue` only requires the caller to hold `QLock` and method `fireEQEvent` only requires the caller to hold `ListLock`. Because there are no direct effects on the regions `Queue` and `Listener` in `dispatchEvent`, the analysis supporting the transformation makes use of the `@requiresLock` annotations on the methods `dequeue` and `fireEQEvent`.

8.2.3 Problems

A moment's thought on the results of shrinking the critical sections in `dispatchEvent` produces the conclusion that `EventQueue` is now broken: the first-in-first-out behavior that made it a queue has been compromised. By moving the calls of `dequeue` and `fireEQEvent` into separate critical sections, we have enabled the following scenario. Assume there are at least two events in an `EventQueue eq`, and that no other threads are concurrently accessing `eq`:

1. One thread invokes `dispatchEvent`, which executes through line 5.
2. Now a second thread invokes `dispatchEvent`. Because the first thread is not in a critical section, the second thread can begin to execute the method, and, in fact, executes the method in its entirety.
3. The first thread resumes execution, completing line 6.

This scenario causes events to be sent to listeners out of order, a violation of the intended first-in-first-out behavior of the queue.

8.2.4 Missing Policy

Our application of the **shrink critical section** transformation did not execute safely because it caused the class to change its behavior in a manner contrary to the intent of the programmer. Clearly its preconditions need to be improved: the transformation must respect the relevant programmer-defined concurrency policies. In this case, the programmer needs to specify a policy that forbids `dispatchEvent` to interleave with itself so that the FIFO nature of the queue is maintained. Now **shrink critical section** will unnest the two critical sections in `dispatchEvent`, but also introduce the usage of a new policy lock to prevent the possibility that the method could interleave with itself. The resulting method implementation is

```
/** @writes All */
public void dispatchEvent() {
  synchronized( this ) {
    Object temp;
    synchronized( queue ) { temp = dequeue(); }
    final Object o = temp;
    synchronized( listeners ) { fireEQEvent( o ); }
  }
}
```

8.3 Evolving a Priority Event Queue

We now consider a concrete example of the role of concurrency policy in transformations by evolving an elaborated version of the `EventQueue` class, shown in Figure 8.3. The `EventQueue` now contains two channels—a priority and a normal channel—each of which behaves as a distinct queue. When the `EventQueue` needs to broadcast an event, it always tries to send a priority event; only if no such event is available is a normal event dequeued. The state of instances of the class is now divided into three subregions: `Listeners`, `Normal`, and `Priority`. The fields of `EventQueue` are distributed among these regions: `listeners` into `Listeners`, `normal` and `numNormal` into `Normal`, and `high` and `numHigh` into `Priority`.

8.3.1 The Initial Policy and `EventQueue`

Figure 8.4 shows class `EventQueue` from Figure 8.3 transformed to be concurrent using **make thread safe**. As with the previous example, a new annotation is introduced declaring that `this` is used as the lock to protect region `Instance`, and all the non-`private` methods are made `synchronized`; `private` methods are annotated with `@requiresLock` annotations. The ini-

```java
/** @region public Listeners
 *  @region public Normal
 *  @region public Priority */
public class EventQueue {
  /** @unshared
   *  @aggregate Instance into Listeners */
  private final List listeners; // List of EQListners
  /** @unshared
   *  @aggregate Instance into Normal */
  private final List normal;    // List of EQEvents
  /** @unshared
   *  @aggregate Instance into Priority */
  private final List high;      // List of EQEvents
  /** @mapInto Normal */
  private int numNormal;
  /** @mapInto Priority */
  private int numHigh;

  public EventQueue() {
    listeners = new ArrayList();
    normal = new LinkedList();
    high = new LinkedList();
    numNormal = 0;
    numHigh = 0;
  }

  /** @writes Listeners */
  public void addEQListener( EQListener l ) {
    listeners.add( l );
  }

  /** @writes Listeners */
  public void removeEQListener( EQListener l ) {
    listeners.remove( l );
  }

  /** @writes All */
  private void fireEQEvent( EQEvent e ) {
    List copy = (List)((ArrayList)listeners).clone();
    for( int i = 0; i < copy.size(); i++ ) {
      final EQListener l = (EQListener)copy.get( i );
      l.dequeued( e );
    }
  }

  /** @reads Normal, Priority */
  public int getSize() {
    int n, h;
    h = numHigh;
    n = numNormal;
    return n + h;
  }

  /** @writes Normal */
  public void enqueue( EQEvent e ) throws NullPointerException {
    if( e == null ) {
      throw NullPointerException( "Cannot enqueue null" );
    } else {
      normal.add( e );
      numNormal += 1;
    }
  }

  /** @writes Priority */
  public void enqueuePriority( EQEvent e ) throws NullPointerException {
    if( e == null ) {
      throw NullPointerException( "Cannot enqueue null" );
    } else {
      high.add( e );
      numHigh += 1;
    }
  }

  /** @writes Normal, Priority */
  private EQEvent dequeue() {
    EQEvent e = dequeuePriority();
    if( e == null ) e = dequeueNormal();
    return e;
  }

  /** @writes Normal */
  private EQEvent dequeueNormal() {
    EQEvent e = null;
    if( numNormal != 0 ) {
      e = (EQEvent)normal.remove( 0 );
      numNormal -= 1;
    }
    return e;
  }

  /** @writes Priority */
  private EQEvent dequeuePriority() {
    EQEvent e = null;
    if( numHigh != 0 ) {
      e = (EQEvent)high.remove( 0 );
      numHigh -= 1;
    }
    return e;
  }

  /** @writes All */
  public void dispatchEvent() {
    EQEvent e = null;
    e = dequeue();
    if( e != null ) fireEQEvent( e );
  }
}
```

Figure 8.3: Initial form of the priority `EventQueue`.

```
/** ...
 *  @lock Lock is this protects Instance */
public class EventQueue { ...
  public EventQueue() { ... }

  /** @writes Listeners */
  public synchronized void addEQListener( EQListener l ) { ... }

  /** @writes Listeners */
  public synchronized void removeEQListener( EQListener l ) { ... }

  /** @writes All
   *  @requiresLock Lock */
  private void fireEQEvent( EQEvent e ) { ... }

  /** @reads Normal, Priority */
  public synchronized int getSize() { ... }

  /** @writes Normal */
  public synchronized void enqueue( EQEvent e )
  throws NullPointerException { ... }

  /** @writes Priority */
  public synchronized void enqueuePriority( EQEvent e )
  throws NullPointerException { ... }

  /** @writes Normal, Priority
   *  @requiresLock Lock */
  private EQEvent dequeue() { ... }

  /** @writes Normal
   *  @requiresLock Lock */
  private EQEvent dequeueNormal() { ... }

  /** @writes Priority
   *  @requiresLock Lock */
  private EQEvent dequeuePriority() { ... }

  /** @writes All */
  public synchronized void dispatchEvent() { ... }
}
```

Figure 8.4: Class `EventQueue` after being made concurrent. Differences from Figure 8.3 are underlined.

tial internal concurrency policy forbids any non-`private` method to interrupt any other. The class is obviously consistent with this policy.

8.3.2 Enabling Liberalized Policy Specification for `EventQueue`

Before the internal concurrency policy of `EventQueue` can be liberalized, the class must be modified to reduce the granularity of the data protections. The **split lock** transformation is applied to the class, so that the regions `Listeners`, `Normal`, and `Priority` are separately protected; the objects referred to by the fields `listeners`, `normal`, and `high` are used as the new locks, respectively. Unlike the preliminary example in Section 8.2, transformation does not remove the `synchronized` declarations from the non-`private` methods. To do so would cause the class to violate its internal concurrency policy. The resulting class is shown in Figure 8.5; notice that `this` has been turned into a policy lock. The scopes of the critical sections generated by the **split lock** transformation are then shrunk by performing a **shrink critical sections** transformation. The results are shown in Figure 8.6.

Now that `EventQueue` has methods with more than one critical section, it is possible to consider how its internal concurrency policy might be liberalized. Here we consider the reasoning about internal policy based on existing state-based critical sections. These are abstracted in Figure 8.7. In particular, we consider how a critical section for region R in one method might observe changes made in a critical section for R in a concurrently executing method, and how this observation may violate higher-level design intentions regarding the behavior of the methods. Methods `getSize` and `dispatchEvent` have potentially interesting interleavings with other methods because they have multiple critical sections. We consider the derivation of two different internal polices, though more are possible: (1) `getSize` always returns a value that is greater than or equal to the actual size of the queue at the time the method returns; and (2) `getSize` always returns a value that is less than or equal to the actual size of the queue at the time the method returns. Both policies insure that events are dispatched in the same order they are enqueued, and that if a normal event is dispatched, the queue does not contain a priority event when `dispatchEvent` returns.

8.3.3 Maximizing `getSize`

The policy matrix describing the maximizing internal concurrency policy is shown in Figure 8.8. The methods `addEQListener` and `removeEQListener` do not affect the contents of the queue, *i.e.*, the regions `Normal` and `Priority`, so their effects on policy development are not discussed. Let us consider explicitly the remaining pairwise cases.

Method `getSize` and itself

Two calls to `getSize` could interleave in one of two ways: (1) $\mathtt{P}_{gs1}$; $\mathtt{P}_{gs2}$; $\mathtt{N}_{gs1}$; $\mathtt{N}_{gs2}$ and (2) $\mathtt{P}_{gs1}$; $\mathtt{P}_{gs2}$; $\mathtt{N}_{gs2}$; $\mathtt{N}_{gs1}$. Because `getSize` only reads data, it cannot interfere with itself, and we thus allow it to execute concurrently with itself.

```
/** @region public Listeners
 *  @region public Normal
 *  @region public Priority
 *  @lock ListLock is listeners protects Listeners
 *  @lock NormalLock is normal protects Normal
 *  @lock HighLock is high protects Priority
 *  @policyLock Serial is this */
public class EventQueue {
  /** @unshared
   *  @aggregate Instance into Listeners */
  private final List listeners; // List of EQListners
  /** @unshared
   *  @aggregate Instance into Normal */
  private final List normal;    // List of EQEvents
  /** @unshared
   *  @aggregate Instance into Priority */
  private final List high;      // List of EQEvents
  /** @mapInto Normal */
  private int numNormal;
  /** @mapInto Priority */
  private int numHigh;

  public EventQueue() { ... }

  /** @writes Listeners */
  public synchronized void addEQListener( EQListener l ) {
    synchronized( listeners ) { listeners.add( l ); }
  }

  /** @writes Listeners */
  public synchronized void removeEQListener( EQListener l ) {
    synchronized( listeners ) { listeners.remove( l ); }
  }

  /** @writes All
   *  @requiresLock ListLock */
  private void fireEQEvent( EQEvent e ) {
    List copy = (List)((ArrayList)listeners).clone();
    for( int i = 0; i < copy.size(); i++ ) {
      final EQListener l = (EQListener)copy.get( i );
      l.dequeued( e );
    }
  }

  /** @reads Normal, Priority */
  public synchronized int getSize() {
    synchronized( high ) {
      synchronized( normal ) {
        int n, h;
        h = numHigh;
        n = numNormal;
        return n + h;
      }
    }
  }

  /** @writes Normal */
  public synchronized void enqueue( EQEvent e ) throws NullPointerException {
    synchronized( normal ) {
      if( e == null ) {
        throw NullPointerException( "Cannot enqueue null" );
      } else {
        normal.add( e );
        numNormal += 1;
      }
    }
  }

  /** @writes Priority */
  public synchronized void enqueuePriority( EQEvent e ) throws NullPointerException {
    synchronized( high ) {
      if( e == null ) {
        throw NullPointerException( "Cannot enqueue null" );
      } else {
        high.add( e );
        numHigh += 1;
      }
    }
  }

  /** @writes Normal, Priority
   *  @requiresLock NormalLock, HighLock */
  private EQEvent dequeue() {
    EQEvent e = dequeuePriority();
    if( e == null ) e = dequeueNormal();
    return e;
  }

  /** @writes Normal
   *  @requiresLock NormalLock */
  private EQEvent dequeueNormal() {
    EQEvent e = null;
    if( numNormal != 0 ) {
      e = (EQEvent)normal.remove( 0 );
      numNormal -= 1;
    }
    return e;
  }

  /** @writes Priority
   *  @requiresLock HighLock */
  private EQEvent dequeuePriority() {
    EQEvent e = null;
    if( numHigh != 0 ) {
      e = (EQEvent)high.remove( 0 );
      numHigh -= 1;
    }
    return e;
  }

  /** @writes All */
  public synchronized void dispatchEvent() {
    synchronized( high ) {
      synchronized( normal ) {
        synchronized( listeners ) {
          EQEvent e = null;
          e = dequeue();
          if( e != null ) fireEQEvent( e );
        }
      }
    }
  }
}
```

Figure 8.5: Class `EventQueue` after a split-lock transformation. Differences from Figure 8.4 are underlined.

```
public class EventQueue { ...
  public EventQueue() { ... }

  /** @reads Normal, Priority */
  public synchronized int getSize() {
    int n, h;
    synchronized( normal ) { h = numHigh; }
    synchronized( high ) { n = numNormal; }
    return n + h;
  }

  /** @writes Normal */
  public synchronized void enqueue( EQEvent e ) throws NullPointerException {
    if( e == null ) {
      throw NullPointerException( "Cannot enqueue null" );
    } else {
      synchronized( normal ) {
        normal.add( e );
        numNormal += 1;
      }
    }
  }

  /** @writes Priority */
  public synchronized void enqueuePriority( EQEvent e ) throws NullPointerException {
    if( e == null ) {
      throw NullPointerException( "Cannot enqueue null" );
    } else {
      synchronized( high ) {
        high.add( e );
        numHigh += 1;
      }
    }
  }

  /** @writes All */
  public synchronized void dispatchEvent() {
    EQEvent e = null;
    synchronized( high ) {
      synchronized( normal ) { e = dequeue(); }
    }
    if( e != null ) synchronized( listeners ) { fireEQEvent( e ); }
  }
}
```

Figure 8.6: Class `EventQueue` after a shrink-critical-sections transformation. Differences from Figure 8.5 are underlined.

Method	*Critical Sections*
`addEQListener`	`L`
`removeEQListener`	`L`
`getSize`	`P;N`
`enqueue`	`N`
`enqueuePriority`	`P`
`dispatchEvent`	`P(N);L`

Figure 8.7: Critical sections accessed by `EventQueue`'s methods in Figure 8.6. `L`, `N`, and `P` represent critical sections for regions `Listeners`, `Normal`, and `Priority`, respectively. Sequential accesses are separated by a '`;`'. Nested accesses are notated within parentheses.

		(1)	(2)	(3)	(4)	(5)	(6)
(1)	`addEQListener`	S					
(2)	`removeEQListner`	S	S				
(3)	`getSize`	S	S	S			
(4)	`enqueue`	S	S	S	S		
(5)	`enqueuePriority`	S	S	×	S	S	
(6)	`dispatchEvent`	S	S	S	S	×	×

Figure 8.8: The internal concurrency policy for maximizing `getSize`.

Methods `getSize` and `enqueue`

A call to `enqueue` could interleave with `getSize` in exactly one way: $\mathtt{P}_{gs}$; $\mathtt{N}_e$; $\mathtt{N}_{gs}$. Because `getSize` would read the number of normal events in critical section $\mathtt{N}_{gs}$ after it is incremented in $\mathtt{N}_e$, it would return the exact size of the queue. This is consistent with our intended semantics for `getSize` so we allow the interleaving.

Methods `getSize` and `enqueuePriority`

A call to `enqueuePriority` could interleave with `getSize` in exactly one way: $\mathtt{P}_{gs}$; $\mathtt{P}_{ep}$; $\mathtt{N}_{gs}$. Because `getSize` would read the number of priority events in $\mathtt{P}_{gs}$ just prior to it being incremented in $\mathtt{P}_{ep}$, which happens *before* the method returns, it would return a size that is smaller than the actual size of the queue at the point in time that the method returns. We decide, therefore, to disallow the possibility of `enqueuePriority` interleaving with `getSize`.

Methods `getSize` and `dispatchEvent`

Assuming a thread is already executing `getSize`, a concurrent call to `dispatchEvent` could interleave with `getSize` in three ways: (1) $\mathtt{P}_{gs}$; $\mathtt{P}_{de}(\mathtt{N}_{gs};\ \mathtt{N}_{de})$; $\mathtt{L}_{de}$, (2) $\mathtt{P}_{gs}$; $\mathtt{P}_{de}(\mathtt{N}_{de})$; $\mathtt{N}_{gs}$; $\mathtt{L}_{de}$, and (3) $\mathtt{P}_{gs}$; $\mathtt{P}_{de}(\mathtt{N}_{de})$; $\mathtt{L}_{de}$; $\mathtt{N}_{gs}$. In all cases, the number of priority events in the queue could be decremented after `getSize` reads the number of priority events, and thus the number of normal events in the queue read by `getSize` would always be exact because it is read after `dispatchEvent` might have altered it. Under these conditions, `getSize` will always return either the exact size or a size greater than the actual size of the queue at the point in time when it returns.

We have considered how `dispatchEvent` could "interrupt" execution of `getSize`. Now we consider how `getSize` could "interrupt" execution of `dispatchEvent`. The possible interleavings are: $\texttt{P}_{de}(\texttt{N}_{de})$; $\texttt{P}_{gs}$; $\texttt{N}_{gs}$; $\texttt{L}_{de}$ and $\texttt{P}_{de}(\texttt{N}_{de})$; $\texttt{P}_{gs}$; $\texttt{L}_{de}$; $\texttt{N}_{gs}$.[1] In both cases, `getSize` will read the sizes of the two portions of the queue after `dispatchEvent` will have made any modifications, and thus `getSize` will always return the exact size of the queue at the point in time it returns.

We decide therefore that the two methods are allowed to interleave.

Methods `dispatchEvent` and `enqueue`

These methods can interleave in two ways: (1) $\texttt{P}_{de}(\texttt{N}_{e};\ \texttt{N}_{de})$; $\texttt{L}_{de}$ and (2) $\texttt{P}_{de}(\texttt{N}_{de})$; $\texttt{N}_{e}$; $\texttt{L}_{de}$.[2] In the first case, the normal event will be enqueue before `dispatchEvent` may attempt to dequeue one. In the second case, because the event is enqueued after the attempt to dequeue one, `dispatchEvent` may return without having dispatched any events even though it is returning when there is an event in the queue. For the purposes of our example, we decide that this is acceptable behavior, so allow the methods to interleave.

Methods `dispatchEvent` and `enqueuePriority`

There is only one possible interleaving: $\texttt{P}_{de}(\texttt{N}_{de})$; $\texttt{P}_{ep}$; $\texttt{L}_{de}$. This scenario can result in the dispatch of a normal event even though a priority event is available so we decide to disallow this interleaving.

Method `dispatchEvent` and itself

Finally, we consider how `dispatchEvent` could interleave with itself. The possible interleavings are: (1) $\texttt{P}_1(\texttt{N}_1)$; $\texttt{P}_2(\texttt{N}_2)$; $\texttt{L}_1$; $\texttt{L}_2$ and (2) $\texttt{P}_1(\texttt{N}_1)$; $\texttt{P}_2(\texttt{N}_2)$; $\texttt{L}_2$; $\texttt{L}_1$. The first trace is allowable, but the second trace would cause an event to be sent out of order, and thus must be disallowed. We, therefore, disallow `dispatchEvent` from interleaving with itself.

8.3.4 Minimizing `getSize`

We now consider how to specify the policy that insures `getSize` will always return a size that is less than or equal to the actual size of the queue at the point in time that `getSize` returns. The policy matrix describing this minimizing policy is shown in Figure 8.9. We describe only the two cases for which the policy decisions are different from the maximizing case.

[1] Critical section $\texttt{N}_{gs}$ cannot interleave with $\texttt{N}_{de}$ nested inside of $\texttt{P}_{de}$, because critical section $\texttt{P}_{gs}$ must execute before $\texttt{N}_{gs}$, and must appear after $\texttt{P}_{de}$ has executed in its entirety.

[2] This is equivalent to $\texttt{P}_{de}(\texttt{N}_{de};\ \texttt{N}_{e})$; $\texttt{L}_{de}$.

		(1)	(2)	(3)	(4)	(5)	(6)
(1)	`addEQListener`	S	—	—	—	—	—
(2)	`removeEQListner`	S	S	—	—	—	—
(3)	`getSize`	S	S	S	—	—	—
(4)	`enqueue`	S	S	S	S	—	—
(5)	`enqueuePriority`	S	S	$\boxed{\text{S}}$	S	S	—
(6)	`dispatchEvent`	S	S	$\boxed{\times}$	S	×	×

Figure 8.9: The policy matrix for minimizing `getSize`. Differences from Figure 8.8 are boxed.

1. The programmer can allow `enqueuePriority` to interrupt `getSize`. As is previously mentioned, there is only one interleaving: $\mathtt{P}_{gs}$; $\mathtt{P}_{ep}$; $\mathtt{N}_{gs}$. Because the number of priority events in the queue is increased after the number of priority events in the queue is read, `getSize` will return a size that is too small at the point in time when it returns. This is consistent with the minimizing policy.

2. The programmer must, however, disallow `dispatchEvent` from interrupting `getSize`. Because it is previously argued that this interruption can cause `getSize` to return a size that is larger than the actual size at the point in time that it returns, this interruption is clearly contrary to the desired minimizing policy.

8.3.5 A Note on Policy Elicitation

The specification of policy is grounded in the programmer's notion of how the program should behave. We expect that, in practice, interactive tools will elicit policy specifications from the programmer by asking precise questions based on existing critical sections and inferred interleaving possibilities about which method interactions are allowed. In this regard, the programmer does not have to worry about the behavior until asked about it by the tool.

8.3.6 Implementing Policy

The process of liberalizing a policy specification does not alter the source code. The **implement policy** transformation is used to update the code to take advantage of a new internal policy. Considering a method-based policy representation, one approach to this transformation is to associate a new lock with each method that is not allowed to interleave with some other method, or more concretely, to associate a new policy lock with each method whose column[3] in the policy matrix contains a "×." If method `m` has a new policy lock associated with it, the bodies of method `m` and of each method `n` that must not interleave with it are wrapped in `synchronized` blocks that acquire the lock associated with `m`. This locking *replaces* any previous policy-based locking.

Figure 8.10 shows `EventQueue` after being transformed to take advantage of the maximizing `getSize` policy. The policy lock `Serial` (formerly represented by `this`) has been replaced by the policy locks `GSLock`, represented by the new field `gsLock`, and `DLock`, represented by `this`, which prevent methods from interleaving with `getSize` and `dispatchEvent`, respectively. The body of method `getSize` is within a block `synchronized` on `gsLock`. This prevents `enqueuePriority`, which also synchronizes on `gsLock`, from interrupting its execution. The body of method `dispatchEvent` synchronizes on `this`, which prevents `dispatchEvent` from interrupting itself. The implementation is still more conservative than policy allows: method `getSize` is allowed to interrupt itself, but the implementation prevents this due to the use of `gsLock` within `getSize`.

8.4 Policy Specification and Transformation (Reprise)

When a class is initially made concurrent, it is assigned a conservative concurrency policy requiring all the methods of class to execute serially. The class will already be in compliance with the policy: all bodies of all its visible methods will be declared `synchronized`. The policy cannot yet be liberalized because each method only contains one critical section on the same shared region. To liberalize the policy, the protection of data within the class needs to be restructured by introducing new shared regions. This restructuring is based on the region hierarchy within the class: finer grained shared regions generally enable interleavings at a finer granularity of computation. Several source-level manipulations assist the programmer with this restructuring. For example, **split lock** distributes locking responsibility across the sub-regions of a formerly singularly protected super-region. Another transformation, **shrink critical section**, uses effects analysis to contract the syntactic scope of a critical section by moving its beginning and end past statements that do not affect the region being protected by the critical section.

The initial definition of regions within the class reflects the programmer's understanding of how fields of the class are related. In particular, concurrency opportunities are introduced or eliminated

[3]We could just as easily associate a lock with a row, the point here is to introduce only one lock for each pair of methods that must not execute interleaved.

```
/** @region public Listeners
 *  @region public Normal
 *  @region public Priority
 *  @lock ListLock is listeners protects Listeners
 *  @lock NormalLock is normal protects Normal
 *  @lock HighLock is high protects Priority
 *  @policyLock GSLock is gsLock
 *  @policyLock DLock is this */
public class EventQueue {
  /** @unshared
   *  @aggregate Instance into Listeners */
  private final List listeners; // List of EQListners
  /** @unshared
   *  @aggregate Instance into Normal */
  private final List normal;    // List of EQEvents
  /** @unshared
   *  @aggregate Instance into Priority */
  private final List high;      // List of EQEvents
  /** @mapInto Normal */
  private int numNormal;
  /** @mapInto Priority */
  private int numHigh;

  protected final Object gsLock = new Object();

  public EventQueue() { ... }

  /** @writes Listeners */
  public void addEQListener( EQListener l ) {
    synchronized( listeners ) { listeners.add( l ); }
  }

  /** @writes Listeners */
  public void removeEQListener( EQListener l ) {
    synchronized( listeners ) { listeners.remove( l ); }
  }

  /** @writes All
   *  @requiresLock ListLock */
  private void fireEQEvent( EQEvent e ) {
    List copy = (List)((ArrayList)listeners).clone();
    for( int i = 0; i < copy.size(); i++ ) {
      final EQListener l = (EQListener)copy.get( i );
      l.dequeued( e );
    }
  }

  /** @reads Normal, Priority */
  public int getSize() {
    synchronized( gsLock ) {
      int n, h;
      synchronized( high ) { h = numHigh; }
      synchronized( normal ) { n = numNormal; }
      return n + h;
    }
  }

  /** @writes Normal */
  public void enqueue( EQEvent e ) throws NullPointerException {
    if( e == null ) {
      throw NullPointerException( "Cannot enqueue null" );
    } else {
      synchronized( normal ) {
        normal.add( e );
        numNormal += 1;
      }
    }
  }

  /** @writes Priority */
  public synchronized void enqueuePriority( EQEvent e ) throws NullPointerException {
    synchronized( gsLock ) {
      if( e == null ) {
        throw NullPointerException( "Cannot enqueue null" );
      } else {
        synchronized( high ) {
          high.add( e );
          numHigh += 1;
        }
      }
    }
  }

  /** @writes Normal, Priority
   *  @requiresLock NormalLock, HighLock */
  private EQEvent dequeue() {
    EQEvent e = dequeuePriority();
    if( e == null ) e = dequeueNormal();
    return e;
  }

  /** @writes Normal
   *  @requiresLock NormalLock */
  private EQEvent dequeueNormal() {
    EQEvent e = null;
    if( numNormal != 0 ) {
      e = (EQEvent)normal.remove( 0 );
      numNormal -= 1;
    }
    return e;
  }

  /** @writes Priority
   *  @requiresLock HighLock */
  private EQEvent dequeuePriority() {
    EQEvent e = null;
    if( numHigh != 0 ) {
      e = (EQEvent)high.remove( 0 );
      numHigh -= 1;
    }
    return e;
  }

  /** @writes All */
  public synchronized void dispatchEvent() {
    EQEvent e = null;
    synchronized( high ) {
      synchronized( normal ) { e = dequeue(); }
    }
    if( e != null ) synchronized( listeners ) { fireEQEvent( e ); }
  }
}
```

Figure 8.10: Final liberalized implementation of `EventQueue`.

according to the programmer's notion of how the different regions of state are related, and the end policy that the programmer intends to specify. For example, in `EventQueue`, we adopted a policy that allows for operations on listeners to execute concurrently with the manipulation of queue items. The protection of listeners, therefore, was separated from the protection of queue elements. Similarly, because we wanted to be able to concurrently enqueue both a normal and a high-priority event, the protection of the two queue segments was made distinct. If we did not need to be able to manipulate both kinds of events simultaneously, there would be no benefit, from the point of view of policy, gained from making distinct regions for normal and high-priority events.

Once the protection of state has been restructured, the policy specification can be liberalized. If a liberal-enough policy is not specifiable, there are two options. The programmer can repeat the process of altering the structure of shared regions to expose more potential concurrency within the methods. If this restructuring is not possible, or the additional concurrency results in behavior that violates the intended behavior of the class, then the programmer may have to reconsider the desired behavior. Such reconsiderations are translated into policy by enabling pairs of methods to interleave that were previously not allowed to interleave.

The internal concurrency policy of a class influences which transformations may be applied to the code: modifications that would enable the class to behave contrary to the policy are meant to be disallowed. *It is important to recognize that in our generative approach the policy evolves along with the program.* A policy might be initially restrictive; as the programmer determines that various transformations should be applied to the program, the policy can be relaxed—if the programmer decides that relaxing the policy is acceptable—to enable them. Once a satisfactory policy is specified, the implementation of the class can be transformed to take advantage of newly allowable concurrency. Of course, it should also be possible to evolve the policy to be more restrictive. This presents more difficulty, and may best be addressed by first retreating to a previous state of development that employs a more restrictive policy than the one desired.

8.5 The Split Lock Transformation

Split lock is parameterized by the `@lock` annotation of the lock to be split, giving a lock L with name M associated with a shared region R. The lock annotations within the class are analyzed to determine all the subregions $R_1, \ldots, R_n$ of R. For each child region 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 F_i for that region.

We first describe the transformation ignoring the existence of `@requiresLock` annotations. All the method and constructor bodies of the class are searched to find all the `synchronized` blocks that use lock L. For each such `synchronized` block, the transformation

- Determines the child regions $\{R_j\}$ of R that are affected by the body. Effects outside of region R are not interesting. The exact child regions of R that are affected can be determined because the exact fields affected are known; any field affecting region R is either a member of a child of R, or is itself a child of R.
- Replaces the `synchronized` block with a set of nested `synchronized` blocks that acquire the appropriate locks $\{L_j\}$ for the regions $\{R_j\}$.

The class is also modified by the addition and removal of annotations:

- The original `@lock` annotation is removed.
- Any annotations `@returnsLock` M are removed—it ought to be the case that any method so annotated no longer has any callsites.
- For each child region R_i of R, a new lock declaration is added to the class

  `@lock` M_i `is` F_i `protects` R_i

8.5.1 Handling `@requiresLock` Annotations

The easiest way to deal with `@requiresLock` annotations is to replace any appearance of M in them with $M_1, \ldots, M_n$. This is always correct, and does not interfere with existing uses of the methods. This would be done before modifications are made to any `synchronized` blocks. This approach, however, can unnecessarily constrain the future use of the methods because they would require more locks than they may actually need. The difficulty in making the annotations less restrictive is *discerning the programmer's intent*: the programmer might want to use less general annotations to preserve flexibility for future uses of the method. Analysis of the implementation of the method can be used to determine the new minimum `@requiresLock` annotations needed for all the methods currently using M. Such an analysis would need to be iterative, because the annotations on one method affect the annotations on the methods that call it. The tool would then present the minimum annotations to the user, who could then enlarge them if desired.

8.5.2 Policy Issues

Currently, policy locks are always acquired before locks associated with regions. So, while any `synchronized` statement replaced by **split lock** that occurs within a `synchronized` statement acquiring a policy lock can affect the interleavings that are exposed when policy is defined, it will not alter the interleavings that could actually occur at runtime. Altered `synchronized` blocks that do not appear within policy-related `synchronized` blocks *can* affect the interleavings that may occur at runtime. For example, after applying **split lock**, two critical sections that formerly acquired a lock L may now acquire locks M and N, respectively. These two critical sections no longer exclude each other, and their concurrent execution may produce effects contrary to the desired concurrency policy. To prevent the current policy from being violated, therefore, a new policy lock needs to be declared in the class and new `synchronized` blocks that acquire it must be placed around any `synchronized` blocks introduced by the manipulation that are not otherwise within policy-related synchronized blocks.

8.6 The Shrink Critical Section Transformation

In its most basic form, **shrink critical section** operates on a single `synchronized` block, limiting its effects to a single method. The transformation changes the syntactic size of a critical section; a different transformation is used to split a single critical section into multiple critical sections. Compound variations of the transformation can be assembled that would shrink all the critical sections of a particular shared region within a method, class, or subtree of the class hierarchy, or that shrink all the critical sections within a method, class, or subtree of the class hierarchy.

Shrink critical section takes as input a single `synchronized` block. The block's expression must be a final expression so that the lock being used can be identified; see Section 5.6. From the lock, the shared region being protected can be determined. This operation cannot be applied to a policy lock.

It seems that this should be an easy transformation to perform: the effects of the statements in the `synchronized` block could be used to move the start and end of the block past statements that do not have effects on the shared region it is protecting. In addition, the `synchronized` block could be moved inside of a compound statement if that statement is the only remaining content of the `synchronized` block. There are many complicating factors, however:

- The order of nested synchronized statements cannot be changed because of deadlock concerns.

- The bounds of a `synchronized` block cannot be moved across looping constructs. This would split the critical section. Introducing new critical sections alters how methods can interleave, and would affect policy.
- `switch` statements are problematic because some `cases` might "fall through" to others, which could also cause the possibility of split critical sections.

In short, the transformation's preconditions, particularly those concerning policy are complicated, and require further research.

8.7 Developing the Generative Approach

The current presentation is of a notional generative approach to concurrency management: we have not yet implemented any transformations. Before we can implement a transformation, the precise details of its specification must be elaborated: what are its preconditions, how does it use analysis and annotation, how does it appeal to the concurrency policy, what exactly does it do to the source code, what are the caveats of a successful transformation, *etc*. In addition, the concurrency-related transformations mentioned so far consider only a single `final` class, and affect only the number, scope, and granularity of critical sections used by its method implementations. These kinds of manipulations relate to how an object can be manipulated by multiple threads. Transformations that introduce the definition and use of threads are missing, as are manipulations that introduce and evolve the use of condition variables. Additional work is needed to understand what form these manipulations should take, and what their effects should be. These techniques can also be extended to transform non-`final` classes.

The generative approach could be expanded to handle the evolution of confederations of instances that cooperate in their use of concurrency; *e.g.*, a class that uses delegates that is intended to appear to be a single monitor. Transformations that can introduce and evolve such relationships need to be identified, but first our understanding of the nature of collaborations needs to be improved.

8.8 Related Work

Several areas of research are related to our proposed transformation-based approach to concurrency. In this section we first discuss Andrews's global invariant approach to the development of correct concurrent programs in which the programmer designs a program first at a coarse granularity of atomicity, then at a finer granularity, and finally converts the design into a specific implementation. We next review previous transformational approaches to converting sequential programs into correct concurrent programs. Finally, we review literature on compiler-based transformations that optimize—generally by attempting to minimize the number of critical sections—the acquisition of locks.

8.8.1 The Global Invariant Approach

Andrews's global invariant approach, introduced in [And89] and elaborated in [And91], is a systematic method for solving synchronization problems based on viewing processes as invariant maintainers. Maintainence of the invariant insures that the processes are interference-free. The method consists of four steps: (1) define the problem, (2) outline a solution, (3) ensure the invariant, and (4) implement the atomic actions. In the first step the processes are identified as sequential algorithms, and the invariant is specified over shared state. This state is often introduced solely to represent explicitly information about the processes that is otherwise unrelated to the underlying computation, such as the number of proceseses currently executing a critical section. Follow up work [Miz01a], discussed below, identifies many common invariant patterns that simplifies this process and frees the programmer from having to introduce variables for such accounting purposes. In the second step, the programmer introduces assignments to the shared variables so that the invariant is initially true and groups assignments into atomic actions.

In the third step, the programmer guards each atomic assignment as necessary to insure that the state resulting from the assignment statisfies the invariant. The necessary guards can be mechanically derived using Dijkstra's weakest precondition. Finally, the in the fourth step, the guarded atomic statements are implemented on top of an underlying synchronization mechanism such as semaphores or monitors. More recently, Mizuno describes how to perform the fourth step for Java [Miz99]. His translation presents a solution to the problem of using multiple condition variables associated with a single mutex in Java, while making efficient use of notification.

The second and third steps are mechanical once the invariant is identified in the first step. Recently published refinements to the process and recently developed tool support make the fourth step mechanical as well. Mizuno, *et al*. [Miz01b] have integrated Andrews's basic method into scenario-based development methologies, such as the Unified Modeling Language and Rational Unified Process, and incorporated it into an aspect-oriented programming [KLM$^+$97] framework. Specification of the invariant and the associated synchronization described in a synchronization aspect distinct from the sequential implementation of processes. Use-case realizations, made up of, for example, sequence diagrams, collaboration diagrams, and scenarios, describe the collaborations between classes and objects in the system. The developer identifies synchronization regions in scenarios. “A synchronization region is a segment in a scenario (1) in which a thread waits for some event to occur or some state to hold ... and (2) in which a thread may trigger an event or change a state for which a thread at some other synchronization region is waiting.” The reflexive transitive closure of the relation between communicating regions defines equivalence classes called clusters. The developer identifies a global invariant for each cluster. Course-grained and fine-grained solutions follow, except that they describe only the entry and exit points to each cluster. A final weaving step integrates the fine-grained synchronization aspect with the component code developed from the scenarios.

The primary difficulty of the global invariant approach is identifying appropriate invariants. Mizuno [Miz01a] describes a useful set of global invariant patterns that can be composed to succintly and more easily specifiy global invariants for clusters. The patterns are all defined in terms of “in” and “out” counters that track the number of threads that have entered and exited a synchronization region.

Deng, *et al*. [DDHM02] implement the aspect-oriented synchronization approach for C++ and Java in the SyncGen tool. The programmer identifies synchronization regions and clusters in the core functional, *i.e*., non-synchronization code, using annotations. The programmer specifies a synchronization invariant for each cluster using conjunctions of invariant patterns. The global invariants are automatically translated into implementation-independent coarse-grained solutions. SyncGen can translate the coarse-grained solution into implementation-specific fine-grained solutions for Java, C, and C++ environments and weave the synchronization code with the core functional implementation. A final model-checking step uses the Bandera framework [CDH$^+$00] to verify the translation process.

8.8.2 Sequential to Concurrent Programs

In Path Pascal [CK79], the programmer writes objects in a purely sequential manner. Allowable concurrency is specified separately, as the class’s path expression. Every method must appear at least once in the path expression, which specifies the relationships among the methods in the class. For example, a relative order of execution among a set of methods can be specified, a set of methods can be made mutually exclusive, or the number of active executions of a given method can be limited. Rules to generate method prologues and epilogues that make use of counting semaphores from the

the path expression are provided. Path expressions also require that the ability of a method to run depend only on what other methods are currently executing or have executed; it can not depend on the current state of the object. The purpose of path expressions is to encapsulate the abstract relationship between methods, separate from the methods' implementations. The preservation of data invariants is accomplished solely through the specification of the path expression for a class; there is no such thing as a critical section, and therefore, there is no way to identify statement-level parallelism.

Wu and Lewis [WL90] apply path expressions to C++ objects to describe the legal concurrent executions of methods of that object. By checking uses of objects of the class against the path expression, illegal concurrent uses of the class can be statically identified. Instead of translating the path expression into code that implements the policy, they use an algorithm based on data dependence information and the path expression to identify regions of code that can be automatically parallelized because it is known that the parallelized code will invoke methods in a manner that respects the path expression.

Sims and Hensgen [SH93] describe how to automatically convert a sequential object to a concurrent object that uses two-phased locking. The concurrent object will be free of race conditions and deadlock. The technique allows for unrelated methods to execute concurrently. Unfortunately, they omit the details of how to determine what locks are needed, and what state the locks are actually protecting.

Herlihy [Her91] describes a technique for transforming a sequential implementation of a data structure into a wait-free and non-blocking concurrent data object. The sequential implementation is not general, but written with the transformation in mind. The non-blocking transformation is based on being able to copy object state. Objects are divided into blocks of state, and the sequential implementation is not allowed to modify anything other than the blocks making up the object. Each operation supported by the object must be well defined for every legal state of the object. For objects that occupy more than one block, efficiency is increased by only copying those blocks of the object that are changed by an operation. This must be done by the implementor who must write each operation in a functional style. Instead of modifying the blocks directly, a method returns a new set of blocks representing the new state of the object. The actual transformation from sequential object to wait-free, non-blocking concurrent object is mechanical, and is intended to be performed by a pre-processor or compiler.

Alemany and Felten [AF92] and Barnes [Bar93] have followed up on Herlihy's work, but are mainly concerned with improving the bounds on the execution time. Barnes's work does allow for concurrent modification of objects under some circumstances, but requires that the original sequential operations be specifically written in such a way that they can cooperate.

8.8.3 Compiler Optimizations

Plevyak, Zhang, and Chien [PZC95] describe optimizations that a compiler can perform on concurrent object-oriented languages. In the runtime system they optimize for, every object has its own lock to implement access control on its own state. They focus on inlining method calls to expose critical sections that can be merged. An inlined method call can execute only if any locks that it requires are not held. Methods identified for inlining, therefore, are replaced with a conditional statement: the true branch contains the inlined method surrounded by code that acquires and releases the necessary locks; the false branch invokes the method normally. The expression that checks if runtime conditions permit inlining together with the true branch of the conditional are known as an *access region*. The focus of their work is the enlargement of access regions to reduce the overhead of critical sections. Access regions are expanded by moving code into the true branch of the conditional, and by merging adjacent access regions; new, empty access regions can also be introduced, and then populated by expansion.

Mutual exclusion is maintained by only expanding access regions to include statements whose locking conditions are subsumed by those of the access region, and by insuring that when storage accesses for the same object from two distinct access regions are moved into the same region, they "occur in whole, before or after each other, ensuring locally the mutual exclusion that the programming model guarantees." Deadlock introduction is prevented by disallowing a statement that may block on a resource from being moved into an access region. Also, when an empty access region is created, all locks are acquired atomically (*i.e.*, either all the locks or none of the locks are acquired); this prevents new resource dependencies, and thus new sources of deadlock, from being introduced.

Diniz and Rinard [DR96] present two compiler optimizations that decrease the granularity of locks used in automatically parallelized object-oriented programs. Every object originally has its own lock, which is acquired whenever data in that object is accessed. *Data lock coarsening*, associates a single lock with a set of objects that are used together. This is similar to using regions with uniqueness aggregation in our system. They avoid the aliasing issue by restricting the use of the classes of nested objects. *Computation lock coarsening* transforms a computation that repeatedly acquires and releases the same lock into a computation that acquires and releases the lock only once. The algorithm they present for data lock coarsening attempts to protect nested objects with the same lock as their enclosing object, and results in the introduction of synchronized and synchronization-free versions of methods. Computation lock coarsening is performed after data lock coarsening, and can be applied to a method if (1) it does not contain concurrency introducing constructs, (2) only accesses data in its receiver and the nested objects of the receiver, and (3) all the methods of nested objects that it invokes are protected by the method's receiver's lock. Computation lock coarsening alters the method so that its first statement acquires the lock, its last statement releases the lock, and its method calls only invoke synchronization-free methods. Data lock coarsening cannot introduce deadlock because the original execution model dictates that no method of enclosing-class C can be called when locks are held, and methods of C will only ever obtain a single lock. Computation lock coarsening does not introduce deadlock because it only replaces repeated acquisitions of the same lock with a single acquisition of the lock for a longer period of time.

In follow-up work [DR97], Diniz and Rinard describe generic techniques for increasing the size of critical sections that replace their previous lock coarsening techniques. Their goal is to remove synchronization by eliminating adjacent mutex acquire and release operations. Acquires and releases are first made adjacent by enlarging the size and scope of critical sections: acquires are moved "upwards" along the control flow graph, while releases are moved "downwards" along the control flow graph. A mutex acquisition can be moved upwards, past a node n that has multiple predecessor nodes. In this case, the acquisition is duplicated across all entries into n, and releases are inserted along all paths out of n except for the one from which the acquisition originated. Similarly, a release can be moved downwards, past a node with multiple successors. The compiler identifies pairs of acquire and release operations to move based on reachability within the control flow graph. Movement and cancellation of acquires and releases are applied in one step based on the intersection of reachability paths in the control flow graph. The process is repeated until it cannot be applied any more; termination is assured because the critical sections can only become bigger. To prevent the introduction of deadlock, a mutex operation is not allowed to be moved past a statement that is already in a critical section. The order in which critical sections are nested is, therefore, never altered.

Chapter 9

Conclusion

The failure to express concurrency-related models impedes programmer understanding, explicit assurance of safety, and the safe evolution of concurrent programs. In particular, the design intent describing intended implementation properties is usually missing, and it is generally impossible to assure that an implementation conforms to the programmer's intent, *e.g.*, is free of race conditions. Experience has shown that it is difficult to safely evolve a program without an explicit expression of these models. In general, current programming languages and processes do not facilitate the capture of program design information in ways that are usable—and adoptable—by practicing programmers. Nor do current tools readily provide the means to assure that captured design information is consistent with source code and vice versa. As a result, design information may be lost or out of sync with the reality of the code, obscuring the implementation properties to be assured.

In this dissertation, we present *a programmer-oriented approach to safe concurrency* in which

- Models of programmer design intent are described using formal program annotations concerning mechanical program properties.
- Consistency between source code and annotated design intent is assured using composable static analyses supported by an iterative programmer-led process.

A dominant design consideration for our approach is adoptability by practicing programmers, which influences the design of our annotations and of our prototype analysis tool. In particular, we consider that, at the present state of capability, it is generally unreasonable to require programmers to explicitly express representation invariants. Instead, we ask the programmer to record design intent in terms of properties the programmer is already concerned about. In addition to being easier for the programmer express, these "mechanical" annotations provide value beyond enabling tool-assisted assurance by answering design-related questions a programmer reading the source code might ask. Specifically, we use annotations to express the following model elements:

- To name and hierarchically organize the state of a program, with aggregates that may span multiple objects.
- To describe which state is affected by a method (or other code segment), and what is the nature of the effects.
- To identify uniqueness of object references, modulo temporary "borrowed" references.
- To associate locks with abstract aggregations of state, and provide names for the locks.
- To delineate responsibility for acquiring locks (*e.g.*, caller *vs.* callee).
- To specify which methods may be executed concurrently.

Adoptability strongly influences the design of analysis approach via the principle of "early gratification." Some assurance can be obtained with minimal or no annotation effort, and additional increments of annotation are rewarded with additional increments of assurance.

9.1 Summary of Contributions

This dissertation is about (1) the consequences of missing design intent, (2) capturing programmer design intent via mechanical program annotations, (3) using static analysis to assure consistency between source code and intent, and (4) doing it in a way that is practicable for working programmers. Concurrency provides an excellent problem domain because concurrent programs have so many unexpressed design commitments that are not local to any particular segment of code, and the consequences of errors in concurrent programs can lead to security and reliability faults. Our overall goals are to improve program quality by enabling analyses that assure the safety of programs (with respect to expressed design intentions), and to enable safe program evolution, eventually through the use of tool-supported program transformations. New technical and engineering results contributed by this dissertation include:

- *Regions* provide flexible encapsulations of object state. Specifically, they provide a hierarchical covering of the notional state of an object, enabling state to be named at different granularities. State aggregations can cross object boundaries by (1) the exploitation of unshared (unaliased) references and (2) the parameterization of classes by regions, enabling *parts* of objects to be owned by other objects.
- We describe an *object-oriented effects system* based on the state named by regions.

- We explicitly *associate locks with regions* to protect shared state. Similar approaches in the literature either lack a state model, or use ownership-based models that are incapable protecting state at fine granularity or incorporating only a portion of an object into another object.
- Our notion of *concurrency policy* enables concise descriptions of allowable method interleavings. In particular, concurrency policy is a surrogate for unstated representations that still allows the programmer to differentiate between "good" and "bad" concurrency. Furthermore, it explicitly expresses to clients of a class its intended concurrent behavior.
- *Static analyses* for assuring the consistency between source code and expressed effect- and concurrency-related design intent.
- The beginnings of *an incremental tool-supported process for inserting, assuring, and exploiting annotations expressing models of state and concurrency properties.* Specifically, we implemented a set of Eclipse plug-ins providing a prototype tool for assuring that Java programs are consistent with their associated concurrency-related models. Our tool goes beyond "bug hunting" by providing both explicit positive and negative assurance results; similar tools in the literature are interested primarily in negative results and are typically incapable of providing positive assurance.
- A notional *generative approach to concurrency management* in which semantics-based, behavior preserving, source-level program transformations are used to evolve the amount of concurrency in a program, allowing the exploration of policy and other design decisions while keeping program complexity manageable. In particular, our hierarchical model of state enables easy exploration of the granularity at which state is protected. Concurrency policy plays a fundamental role in preventing transformations from modifying the programmer's design intent with respect to consistency of state.

9.1.1 Case Studies

Throughout, we apply our annotation and assurance techniques to examples drawn from production Java code to experiment with describing design intent, providing assurance of model–code consistency, and finding bugs. In addition to demonstrating the applicability of our approach to production Java code, our case studies with our prototype assurance tool in Eclipse demonstrate the incrementality of our approach, and provide informal evidence that we are providing "early gratification." For example,

- We capture the models of state and locking intent for the `BoundedFIFO` class from the Jakarta Log4j library. Using our prototype assurance tool, we are able to assure that (1) the implementation of `BoundedFIFO` and (2) the implementation of a client of `BoundedFIFO` are consistent with these models. No modifications to the original source code were required beyond the introduction of our annotations. We show how we can express both the internal and external concurrency policies for the class. In addition, we demonstrate notionally how the design intent captured by our annotations can be exploited to preserve the safety of the class during an actual evolution scenario.
- We express the state aggregation and locking intent for the pair of classes `ThreadCache` and `CachedThread` from the W3C Jigsaw web server. In particular, the complexities of state in this example cannot be expressed by similar approaches and tools in the literature.
- We show how to derive and express the internal and external concurrency policies of the class `AppenderAttachableImpl` from Log4j, uncover a policy mismatch error between the class and its clients, and describe how a notional tool could assist in the discovery of this error. This error was reported to the maintainers of the library and resulted in a bug fix.
- We capture the models of state and locking intent for the `Logger` class from the Java JDK `java.util.logging` package. Using our prototype tool, we incrementally and interactively introduce annotations into the source code, receiving additional assurance results at each step. This process required hypothesizing about actual programmer design intent, and making minor modifications to the source code. We discover a previously unreported race condition, and identify several areas of questionable design intent.

More generally, an inspection of the JDK packages `java.util`, `java.lang`, and `java.awt.*`, and `java.io` indicates that our tool and techniques are sufficient to find and explain known concurrency errors: both race conditions and policy violations.

9.2 Looking Forward

Possible future directions for this work include: expressing and assuring additional concurrency-related design intent, expressing patterns of concurrency, incorporating diagrammatic models of intent, and annotation management.

9.2.1 Additional Concurrency-Related Models

This dissertation focuses on concurrency-related models related to shared-memory concurrency programming, in particular models related to the integrity of state. Specific concurrency concerns not addressed in this work include reader–writer locks, deadlock, condition variables, and thread identification.

Greater amounts of concurrency can be obtained by using reader–writer locks, a form of mutex that distinguishes between read and write operations. Multiple threads can acquire the lock to obtain read permission while excluding any threads that want to acquire write access, while exactly one thread at a time can acquire the lock to obtain write access. Java does not natively support reader–writer locks, but objects providing such functionality can be written and are provided by most concurrency utility libraries, *e.g.*, [Lea]. Reader–writer locks present several additional assurance challenges. We must assure that critical sections intended to only read state do not modify the state. Our effects system makes this straightforward, modulo aliasing concerns. Because lock acquisition and release are based on distinguished method calls, we must assure that locks are properly released at the close of critical sections. In general, this is an object protocol problem, *e.g.*, [Nie93], but a specialized analysis for the particular situation is straightforward to implement, *cf.* Warlock [Ste93]. We already describe how concurrency policy can describe reader–writer design intent.

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. A more challenging ordering problem is ordering the instances of a class: this is important, for example, when two instances must be compared. Lea presents a programming idiom based on unique resource identifiers to address this problem [Lea00]. Boyapati, *et al.* present a type system that enforces a resource ordering among nodes in tree structures, and that can even handle dynamic reordering of nodes in the tree [BLR02]. Providing and *assuring* correct use of resource orderings among instances of the same class remains an open problem in general.

We have begun to experiment with the expression of design intent related to condition variables. In [NGS01], we describe how conditions might be explicitly associated with the variable used to control blocking.

Locking is not the only technique for protecting data in concurrent programs: for example, immutable objects can be shared without coordination and thread-local objects are not shared at all. While our notion of concurrency policy is capable of describing some of the design intent embodied in such classes, our current techniques are not sufficient for assuring that an implementation of an object is immutable, or that a reference does not escape a particular thread. Assuring immutability is an application of our state model and effects system. Additional aliasing assurances can assist in the problem of thread-locality. In addition, we are exploring "thread coloring" techniques that make explicit the relationship between threads, data, and code segments [SGS02]. Making thread models explicit facilitates assurance of code and data thread-locality, as well as the identification of shared usage of state *not intended* to be shared.

9.2.2 Concurrency Coding Idioms

Concurrency is often used in a stylized manner, that is, by making use of particular coding patterns, such as those of [Lea00]. The burden of annotating concurrent programs could be lightened by developing annotations that describe a program's use of concurrency at a higher semantic level that accounts for these programming idioms. Examples include a single annotation that declares the intent that a class is intended to be a monitor, or an annotation describing the intent that a particular class is meant to be thread-safe wrapper around another class.

9.2.3 Diagrammatic Models

Models of program behavior can be expressed diagrammatically. We have begun to develop diagrams that capture and express concurrency-related design in terms of mechanical program properties [NGS01]. Such models are an *alternative* means of expressing and maintaining programmer design intent—assurance between annotations, source code, and diagrams must be provided. For example, Figure 9.1 describes how concurrency is used by the class `WakeupManager` in Sun's Jini library, including how state is protected, how locks are ordered, what are the conditions used by the class, and which locks and conditions are used by various methods. `WakeupManager` is a queue of time-stamped tasks sorted by deadline. Nested class `Kicker` implements a thread responsible for executing tasks in the queue and requires the locks for both itself and the queue so lock ordering is used: `ContentsLock` must be acquired before the `KickerLock`. `Kicker` has two condition variables (stop signs in the diagram): one to abort, and one to signal the addition of a new task. From the diagram it is easy to see (1) what is the lock orde,r as well as (2) that invoking `newTime` may cause a thread blocked in `run` to unblock by satisfying the `newTicket` condition.

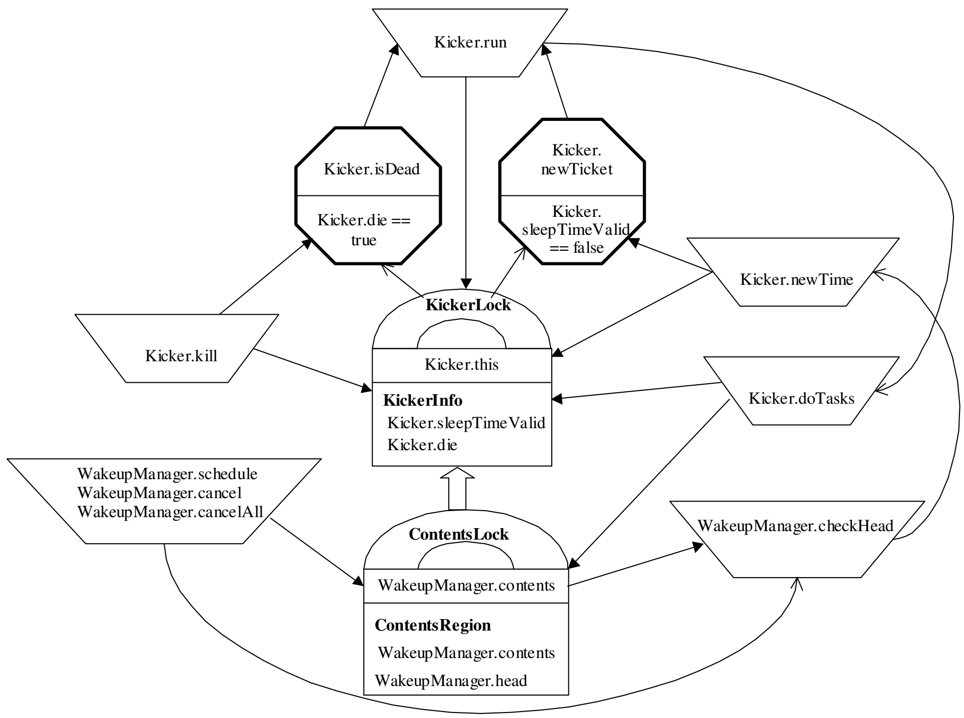


Figure 9.1: Method concurrency diagram for class `WakeupManager`. Diagram by Elissa Newman.

9.2.4 Tool Development

Development of our prototype assurance tool is ongoing. Active areas of research include issues of tool–programmer interaction and truth maintenance of assurance results. In particular, we wish to present to the user "chains of evidence" that link together expressed design intent, analysis results, and code segments to document the reasoning behind each assurance result, positive or negative. While our current case studies suggest that we are on-track towards our goal of providing useful incremental results and "early gratification," formal user studies are required to demonstrate this property more formally.

Bibliography

[AF92] Juan Alemany and Edward W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *PODC '92, Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, pages 125–134. ACM Press, August 1992.

[AFL95] Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 174–185. ACM Press, June 1995.

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. The Java Series. Addison-Wesley, third edition, 2000.

[AM77] G. R. Andrews and J. R. McGraw. Language features for process interaction. In Davd B. Wortman, editor, *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 114–127. ACM Press, March 1977.

[And89] Gregory R. Andrews. A method for solving synchronization problems. *Science of Computer Programming*, 13(1):1–21, December 1989.

[And91] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. The Benjamin/Cummings Publishing Company, Inc, 1991.

[Apa] Apache Software Foundation. Log4j project. `http://jakarta.apache.org/log4j/docs/index.html`.

[Ass98] Association for Computing Machinery. *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*. ACM Press, October 1998.

[Ass99] Association for Computing Machinery. *OOPSLA'99 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*. ACM Press, November 1999.

[Ass00] Association for Computing Machinery. *OOPSLA'00 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*. ACM Press, October 2000.

[Ass02a] Association for Computing Machinery. *OOPSLA'02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*. ACM Press, November 2002.

[Ass02b] Association for Computing Machinery. *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*. ACM Press, June 2002.

[Ass02c] Association for Computing Machinery. *Proceedings of the IEEE International Conference on Software Engineering (ICSE '02)*. ACM Press, May 2002.

[Bar93] Greg Barnes. A method for implementing lock-free shared data structures. In *SPAA '93, Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270. ACM Press, June/July 1993.

[BFC95] Peter A. Buhr, Michael Fortier, and Michael H. Coffin. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, March 1995.

[BG99] John Boyland and Aaron Greenhouse. MayEqual: A new alias question. Presented at IWAOOS '99: Intercontinental Workshop on Aliasing in Object-Oriented Systems. `http://cuiwww.unige.ch/~ecoopws/iwaoos/papers/papers/greenhouse.ps.gz`, June 1999.

[BH74] Per Brinch Hansen. A programming methodology for operating system design. In Jack L. Rosenfeld, editor, *Information Processing 74*, number 6 in IFIP Congress Series, pages 394–397. Elsevier, August 1974.

[BH75] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE-1(2):199–207, June 1975.

[BH99a] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *OOPSLA'99 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications* [Ass99], pages 35–46.

[BH99b] Per Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, April 1999.

[Bir91] A. D. Birrell. An introduction to programming with threads. In Nelson [Nel91], chapter 4, pages 88–118.

[Bla99] Bruno Blanchet. Escape analysis for object-oriented languages: application to Java. In *OOPSLA'99 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications* [Ass99], pages 20–34.

[Blo01a] Joshua Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, 2001.

[Blo01b] Joshua Bloch. Proposed final draft, JSR-41: A simple assertion facility for the Java programming language. `http://jcp.org/jsr/detail/41.jsp`, June 2001.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. A type system for preventing data races and deadlocks in Java programs. In *OOPSLA'02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications* [Ass02a], pages 211–230.

[BNR01] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalization of uniqueness and read-only. In Jørgen Lindskov Knudsen, editor, *ECOOP'01 — Object-Oriented Programming, 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 2–27. Springer, June 2001.

[Boy01a] John Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, May 2001.

[Boy01b] John Boyland. The interdependence of effects and uniqueness. Presented at Workshop on Formal Techniques for Java Programs., June 2001.

[Boy03a] John Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium 2003*. Springer, 2003. To appear.

[Boy03b] John Tang Boyland. Connecting effects and uniqueness with adoption. Submitted to International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming at *ECOOP 2003*, 2003.

[BP03] G. M. Bierman and M. J. Parkinson. Effects and effect inference for a core Java calculus. In *Workshop on Object-Oriented Developments, co-located with ETAPS 2003*, April 2003.

[BR01] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *OOPSLA'01 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, pages 56–69. ACM Press, November 2001.

[BRJ99] Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[BST00] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of Java without data races. In *OOPSLA'00 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications* [Ass00], pages 382–400.

[CBS98] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE '98)*, pages 167–176. IEEE Computer Society, April 1998.

[CD02] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation, and the disjointness of type and effect. In *OOPSLA'02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications* [Ass02a], pages 292–310.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE '00)*, pages 762–765. ACM Press, June 2000.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *OOPSLA'99 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications* [Ass99], pages 1–19.

[CH74] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In E. Gelenbe and C. Kaiser, editors, *Operating Systems; Proceedings of an International Symposium*, volume 16 of *Lecture Notes in Computer Science*, pages 89–102. Springer, April 1974.

[CK79] Roy H. Campbell and Robert B. Kolstad. Path expressions in Pascal. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE '79)*, pages 212–219. IEEE Computer Society, September 1979.

[Cor00] James C. Corbett. Using shape analysis to reduce finite-state models of concurrent Java programs. *ACM Transactions on Software Engineering and Methodology*, 9(1):51–93, January 2000.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications* [Ass98], pages 48–64.

[DDHM02] Xianghua Deng, Matthew B. Dwyer, John Hatcliff, and Masaaki Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE '02)* [Ass02c], pages 442–452.

[DF01] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 59–69. ACM Press, June 2001.

[Dij68a] Edsger W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, Inc., 1968.

[Dij68b] Edsger W. Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[Dis71] Discussion. Monitors—special discussion. In Hoare and Perrot [HP71], pages 72–93.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, Palo Alto, California, USA, December 1998.

[DR96] Pedro Diniz and Martin Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. In *Ninth International Workshop, Languages and Compilers for Parallel Computing*, pages 285–299, August 1996.

[DR97] Pedro Diniz and Martin Rinard. Synchronization transformations for parallel computing. In *Conference Record of the Twenty-fourth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 187–200. ACM Press, January 1997.

[EGHT94] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 87–96. ACM Press, December 1994.

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, January/February 2002.

[FA99a] Cormac Flanagan and Martín Abadi. Object types against races. In Jos C. M. Baeten and Sjoule Maw, editors, *CONCUR '99—10th International Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 288–303. Springer, August 1999.

[FA99b] Cormac Flanagan and Martín Abadi. Types for safe locking. In S. Doaitse Swierstra, editor, *ESOP'99 — Programming Languages and Systems, 8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108. Springer, March 1999.

[FD02] Manuel Fähndrich and Robert DeLine. Adoption and foucs: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation* [Ass02b], pages 13–24.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 219–232. ACM Press, June 2000.

[FF01] Cormac Flanagan and Stephen N. Freund. Detecting race conditions in large programs. In *2001 ACM SIGPLAN–SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 90–96. ACM Press, June 2001.

[FJL01] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 77(2–4):97–108, February 2001.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of the Twenty-fifth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, January 1998.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In José Nuno Oliveira and Pamela Zave, editors, *Formal Methods Europe 2001: Formal Methods Increase Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, March 2001.

[FLL$^+$02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation* [Ass02b], pages 234–245.

[FQ03a] Cormac Flanagan and Shaz Qadeer. Types for atomic interfaces. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, June 2003. To appear.

[FQ03b] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 1–12. ACM Press, January 2003.

[Gai85] Jason Gait. A debugger for concurrent programs. *Software—Practice and Experience*, 15(6):539–554, June 1985.

[GB99] Aaron Greenhouse and John Boyland. An object-oriented effects system. In Rachid Guerraoui, editor, *ECOOP'99 — Object-Oriented Programming, 13th European Conference*, volume 1628 of *Lecture Notes in Computer Science*, pages 205–229. Springer, June 1999.

[GJLS87] D. K. Gifford, P. Jouvelot, J. M. Lucassen, and M. A. Sheldon. FX-87 reference manual. Technical Report MIT/LCS/TR-407, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachussetts, USA, September 1987.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley, second edition, 2000.

[GL86] David K. Gifford and John M. Lucassen. Integrating fuctional and imperative programming. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 28–38. ACM Press, 1986.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. ACME: An architectural description interchange language. In *The IBM Centre for Advanced Studies Conference, CASCON '97*, pages 169–183, November 1997.

[Gre01] Aaron Greenhouse. Bug 1507: Improper synchronization in AsyncAppender and Category causes race conditions. `http://nagoya.apache.org/bugzilla/`, April 2001.

[GS02] Aaron Greenhouse and William L. Scherlis. Assuring and evolving concurrent programs: Annotations and policy. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE '02)* [Ass02c], pages 453–463.

[Ham01] Graham Hamilton. Proposed final draft, JSR-47: Logging API specification. `http://jcp.org/jsr/detail/47.jsp`, September 2001.

[Har98] Stephen J. Hartley. *Concurrent Programming: The Java Programming Language*. Oxford University Press, 1998.

[Hav68] J. W. Havender. Avoiding deadlock in multitasking systems. *IBM Systems Journal*, 7:74–84, 1968.

[Her91] Maurice Herlihy. A methodology for implementing highly concurrent data objects. Technical Report CRL 91/10, Cambridge Research Laboratory, Digital Equipment Corporation, Cambridge, MA, October 1991.

[Hoa71] C. A. R. Hoare. Towards a theory of parallel programming. In Hoare and Perrot [HP71], pages 61–71.

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hol00] Allen Holub. *Taming Java Threads*. Apress, Berkeley, California, USA, 2000.

[How76a] John H. Howard. Proving monitors. *Communications of the ACM*, 19(5):273–279, May 1976.

[How76b] John H. Howard. Signaling in monitors. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE '76)*, pages 47–52. IEEE Computer Society, October 1976.

[HP71] C. A. R. Hoare and R. H. Perrot, editors. *Operating Systems Techniques*. Academic Press, Inc., 1971.

[Hyd99] Paul Hyde. *Java Thread Programming*. Sams Publishing, 1999.

[Jac95] Daniel Jackson. Aspect: Detecting bugs with abstract dependencies. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.

[JG91] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Conference Record of the Eighteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 303–310. ACM Press, January 1991.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-orient programming. In Mehmet Akşit and Stoshi Matsuoka, editors, *ECOOP '97 — Object Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[Lam80] Leslie Lamport. The 'Hoare logic' of concurrent programs. *Acta Informatica*, 14(1):21–37, 1980.

[LB98] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Pthreads*. The Sun Microsystems Press, A Prentice Hall Title, 1998.

[LB99] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Java Technology*. The Java Series. The Sun Microsystems Press, A Prentice Hall Title, 1999.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[Lea] Doug Lea. Overview of package util.concurrent. `http://gee.cs.oswego.edu/dl/`. Version 1.3.1.

[Lea00] Doug Lea. *Concurrent Programming in Java*. The Java Series. Addison-Wesley, second edition, 2000.

[Lei98] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications* [Ass98], pages 144–153.

[LG88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 47–57. ACM Press, January 1988.

[Lip75] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[Lis77] Andrew Lister. The problem of nested monitor calls. *Operating Systems Review*, 11(3):5–7, July 1977.

[LNS00] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Note 2000-002, Compaq Systems Research Center, Palo Alto, California, USA, October 2000.

[Löh93] Klaus-Peter Löhr. Concurrency annotations for reusable software. *Communications of the ACM*, 36(9):81–89, September 1993.

[LP85] Carol H. LeDoux and D. Stott Parker, Jr. Saving traces for Ada debugging. In John G. P. Barnes and Gerald A. Fisher, Jr., editors, *Ada In Use: Proceedings of the Ada International Conference*, The Ada Companion Series, pages 97–108. Cambridge University Press, May 1985. Special edition of *Ada Letters*, Volume V, Issue 2.

[LPHZ02] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation* [Ass02b], pages 246–257.

[Luc87] John M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. PhD thesis, MIT, Cambridge, Massachussetts, USA, September 1987.

[Luc90] David C. Luckham. *Programming with Specifications: An Introduction to ANNA, a Language for Specifying Ada Programs*. Texts and Monographs in Computer Science. Springer, 1990.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[Miz99] Masaaki Mizuno. A structured approach for developing concurrent programs in Java. *Information Processing Letters*, 69:233–238, 1999.

[Miz01a] Masaaki Mizuno. A pattern-based methodology to develop concurrent programs—part 2. Technical Report 2001-03, Department of Computing and Information Sciences, Kansas State University, 2001.

[Miz01b] Masaaki Mizuno. A pattern-based methodology to develop concurrent programs—part 1. Technical Report 2001-02, Department of Computing and Information Sciences, Kansas State University, 2001. Extended and corrected version of [MSN00].

[MSN00] Masaaki Mizuno, Gurdip Singh, and Mitchell Neilsen. A structured approach to develop concurrent programs in UML. In Andy Evans, Stuart Kent, and Bran Selic, editors, *«UML»2000 — The Unified Modeling Language, Third International Conference*, volume 1939 of *Lecture Notes in Computer Science*, pages 451–465. Springer, October 2000.

[NE02] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 11–20. ACM Press, November 2002.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall Series in Innovative Technology. Prentice Hall, 1991.

[NGS01] Elissa Newman, Aaron Greenhouse, and William L. Scherlis. Annotation-based diagrams for shared-data concurrency. In *Workshop on Concurrency Issues in UML*, 2001. Workshop at the Fourth International Conference on the Unified Modeling Language, UML 2001.

[NHP00] James Noble, David Holmes, and John Potter. Exclusion for composite objects. In *OOPSLA'00 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications* [Ass00], pages 13–28.

[Nie93] Oscar Nierstrasz. Regular types for active objects. In Andreas Paepcke, editor, *OOPSLA'93 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, pages 1–15. ACM Press, September 1993.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.

[Ous96] John K. Ousterhout. Why threads are a bad idea (for most purposes). In *1996 USENIX Technical Conference*. USENIX Association, January 1996. `http://home.pacbell.net/ouster/threads.ppt`. No written paper corresponds to this talk.

[OW99] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly & Associaties, second edition, 1999.

[PZC95] John Plevyak, Xingbin Zhang, and Andrew A. Chien. Obtaining sequential efficiency for concurrent object-oriented languages. In *Conference Record of the Twenty-second Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 311–321. ACM Press, January 1995.

[Rey78] John C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, pages 39–46. ACM Press, January 1978.

[Rou03] Vladimir Roubtsov. The thread threat: Why is coding for thread safety still a challenge? *JavaWorld*, February 2003.

[SGS02] Dean F. Sutherland, Aaron Greenhouse, and William L. Scherlis. The code of many colors: Relating threads to code and shared state. In Matthew B. Dwyer, editor, *2002 ACM SIGPLAN–SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 77–83. ACM Press, November 2002.

[SH91] Charles Simonyi and Martin Heller. The Hungarian revolution. *Byte*, 16(8):131–138, August 1991.

[SH93] David L. Sims and Debra A. Hensgen. Automatically mapping sequential objects to concurrent objects: The mutual exclusion problem. In Alok N. Choudhary and P. Bruce Berra, editors, *1993 International Conference on Parallel Processing, Volume II Software*, pages 269–272. CRC Press, Inc., August 1993.

[SP03] Fausto Spoto and Erik Poll. Static analysis for JML's assignable clauses. In *Workshop on Foundations of Object-Oriented Languages (FOOL 10)*, January 2003. `http://www.cis.upenn.edu/~bcpierce/FOOL/papers10/spotopoll.ps`.

[SS84] Peter M. Schwarz and Alfred Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.

[Ste93] Nicholas Sterling. WARLOCK — a static data race analysis tool. In *Proceedings of the Winter 1993 USENIX Conference*, pages 97–106. USENIX Association, January 1993.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the Twenty-third Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 32–41. ACM Press, January 1996.

[TD97] S. Tucker Taft and Robert A. Duff, editors. *Ada 95 Reference Manual: Language and Standard Libraries; International Standard ISO/IEC 8652:1995(E)*, volume 1246 of *Lecture Notes in Computer Science*. Springer, 1997.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of the Twenty-first Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 188–201. ACM Press, January 1994.

[WL90] Youfeng Wu and Ted G. Lewis. Parallelism encapsulation in C++. In David A. Padua, editor, *1990 International Conference on Parallel Processing, Volume II Software*, pages 35–42. The Pennsylvania State University Press, August 1990.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA'99 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications* [Ass99], pages 187–206.

[Yat99] Bennett Norton Yates. A type-and-effect system for encapsulating memory in java. Master's thesis, The Graduate School of the University of Oregon, 1999.