CHARTER

Critical and High Assurance Requirements
Transformed through Engineering Rigor

Project Number ARTEMIS-2008-1-100039

D6.3 JML Model for Realtime Concurrency

Version 1.0
14th June 2012

ARTEMIS JU Distribution

Lero

Project Partners: Aicas, Atego, Chalmers University of Technology, Impronova, Lero, Luminis, NLR, QRTech, Radboud University Nijmegen, The Open Group, University of Twente

This document is deliverable D6.3 for the CHARTER project.

Document Control

Version	Status	Date
0.1	First Prototype	01 May 2011
0.2	Detailed Outline	19 October 2011
0.3	First Draft	January 2012
0.98	Penultimate Draft	13 July 2012
1.0	Final Draft	14 July 2012

Contents

Executive Summary

This deliverable reports on the outcomes of Task 6.3: Model-based Reasoning about Concurrency. The goal of this task was to develop techniques to specify and reason about concurrent RTSJ programs at the model level.

The portion of the work reported in this deliverable resulted in:

- a survey of the evolution of concurrency in the Java platform,
- a review of the main APIs that support programming concurrent and parallel Java programs,
- an analysis of the existing specification constructs that focus on concurrency in the Java Modeling Language and other static analysis tools,
- a survey of existing tools used to reason about concurrency in Java,
- a very large case study which contains empirical analysis of how Java developers in the real world use concurrency,
- the development of a new set of concurrency annotations, available at several refinements levels from architecture models down to program code, and
- a set of JML-annotated concurrency classes, JML model types, and a sketch of a HOL theory of Java concurrency.

This document summarizes these results.

Chapter 1

Introduction

As part of the CHARTER working package WP6 *'Formal Verification'*, the Task 6.3, named *'Model-based Reasoning about Concurrency'*, aims at reasoning about multitasking programs written in Real-time Java and/or Safety-critical Java at the model level.

Task 6.3 was started in months 1–6 then continued, after significant interruption, in months 23–36.

Task 6.3 focuses on: (i) understanding how Java programmers use the modern Java concurrency API (as found in the Java package `java.util.concurrent` and its sub-packages) and (ii) developing a model-based theory of that API.

Consequently, Task 6.3 culminates in deliverable D6.3's analysis of modern Java developers' use of concurrency, via a very large-scale analysis of existing concurrent code, and the *JML Model for Realtime Concurrency*, i.e., a set of JML models formalizing key concurrency constructs in Real-time Java and Safety-Critical Java.

This document is the report for that deliverable, accompanying the model-based specification of the relevant APIs, user documentation, and explanations of this model's characteristics and the underlying analysis that lead to the model's design.

The result of this work (the *JML Model for Realtime Concurrency* itself) complements the subsequent deliverable of Task 6.3, deliverable D6.4, *'A Concurrency Checker'*. The code name of this checker is *VerCors tool*.

Deliverable D6.3 also includes a JML specification of a subset of the modern Java concurrency API.

The scope of this document is to describe the motivations, architecture, design, and intended use of the *JML Model for Realtime Concurrency*. All topics in this introduction will be explained more extensively in the rest of the document.

Chapter 2

Concurrency in Java

A brief summary of the history and current state-of-affairs of concurrency in Java is warranted. This framing is necessary to contextualize the practical, industrial use of concurrency as well as to help chart the evolution of reasoning about concurrent Java within the research community.

2.0.1 Related Books

Several books describe concurrency in Java.

The Java Language Specification, versions 1 through 3 by Gosling, Joy, Steele, and Bracha [37, 38, 39, 40], describe the Java language and its core APIs in detail, including the core constructs supporting concurrency discussed in the next section. Likewise, the four editions of *The Java Programming Language* by Arnold, Gosling, and Holmes (as of the fourth edition) cover this material [5]. A fifth edition of this text is coming out in 2012.

Threads and the Java concurrency API are discussed in detail in the following excellent texts:

- Oaks and Wong in three editions of *Java Threads* [80], the latest of which covers J2SE 5.0,
- Lea's *Concurrent Programming in Java* [63],
- Magee and Kramer's *Concurrency: State Models & Java Programs* [72], and
- Goetz et al.'s *Java Concurrency in Practice* [36].

Magee and Kramer's book is more of an academic textbook than a pedagogical or reference text, as is the case with the other aforementioned books.

Real-time Java is covered in detail mainly by the RTSJ 1.0.2 standard, whose final release 3 was published as a result of JSR-1 in 2006 [11].

Programming against this standard is the focus of Wellings' very good text, *Concurrent and Real-Time Programming in Java* [92].

2.1 Primitive Concurrency in Java Standard Edition

The first release of Java in 1995 included threads as a core construct of the language design. This differentiated Java from many of the other languages in use at the time.

All Java versions up to and including Java 1.4, released in 2003, included only what one might call "primitive" concurrency support.

Concurrency focused on a first-class notion of a thread and a thread group, built-in thread scheduling with no user control, and basic, low-level data synchronization and thread notification.

These latter capabilities were only available via two language keywords, `synchronized` and `volatile`, and through the use of the API of the `java.lang.Object` class.

From the first release of Java, two other classes in the `java.lang` package provide some mechanisms for implementing primitive concurrent programs without threads. The `Process` class is used to represent native operating system processes. In Java versions up to Java 1.4, the `Runtime` class was used to create processes via its `exec` methods and control the virtual machine. A helper class called `ProcessBuilder` was introduced in Java 5.0 to facilitate the creation and control of processes, effectively replacing the process-related facilities found in `Runtime`.

In essence, the key concepts in Java's primitive concurrency support are *threads*, *locks*, *monitors*, and *data synchronization barriers*.

2.1.1 Evolution of Concurrency-related Constructs in Java 1.0 through 1.4

In the early releases of Java in the mid-1990s, Java threads were "green" threads implemented via user-land threads in the virtual machine. The availability of green threads meant that all versions of early Java virtual machines supported "virtual" concurrency whether or not the underlying operating system did so. Moreover, all ports basically behaved in the same fashion, so understanding the behavior of concurrent Java programs across platforms was predictable.

Around the release of Java 1.3 (depending upon the port variant), mapping Java threads to native operating system threads became possible. Many virtual machines of that generation gave developers the option of realizing Java threads as either green threads or native threads for a given execution of the VM. Solaris 7 and onward provided a variety of thread types and a high degree of control over the mapping to, and use of, operating system threads in the Sun VM. Linux's initial support for operating system-level threads came in Java 1.3, and Linux's Native POSIX Thread Library (NPTL) was supported in Java 1.4. Windows received native thread support around this time as well.

Unfortunately, introducing support for native threads meant that the Java's "write once, run anywhere" promise came under increased criticism. In particular, the under-specification of thread scheduling, and the over-promising of thread control, both in Chapter 17 of the JLS and in `java.lang.Thread`'s API were enormous problems. The evolution of latter is quite evident in the deprecation of the methods `stop`, `suspend`, `resume`, and `destroy` in Java 1.2.

In the end, developers could not rely upon any predictable behavior for thread scheduling or control, beyond promises concretized via the `synchronize` and `volatile` keywords.

With regards to these constructs, even their meaning evolved over time. While the `volatile` keyword existed in the Java 1.0 grammar, it was given no formal meaning. Only when Java 1.2 was introduced, concurrent with the publication of the first edition of the JLS, was `volatile`'s meaning fixed. Its meaning evolved again in Java 5.0 with the introduction of the Java Memory Model (discussed later) to that which it is today.

Two other aspects relevant to concurrency were introduced in early Java versions.

First, the `java.lang.ThreadLocal` class was introduced in Java 1.2 to support the idea that each thread should have its own independently initialized copy of a variable. Unfortunately, the performance of `ThreadLocal`s is so poor in many VMs that they were not used but for rare circumstances.

Second, non-blocking I/O was introduced in Java 1.4 via the `java.nio` package and its subpackages. All I/O in Java prior to this release, realized via the `java.io` package, was blocking, and therefore non-blocking I/O had to be simulated through the use of threads.

2.1.2 Criticisms and Weaknesses

Early versions of Java were criticized by many of the leading researchers in language-based concurrency, including the creators of the various concepts that Java adopted.

For example, Per Brinch Hansen, coinventor of the monitor (with Tony Hoare), delivered a withering critique of the synchronization features of Java in "Java's Insecure Parallelism" in 1999 [44]. He concludes that, "Java ignores the last twenty-five years of research in parallel programming languages."

Harold Thimbleby published "A Critique of Java", also in 1999 [88]. A concise summary of his criticism is captured in the quote, "...the language definition should have been an integral part of the design process rather than, as appears, a retrospective commentary."

Additionally, the difficult work in documenting and reasoning about the Java Memory Model (JMM), accomplished by Manson, Pugh, and Adve in 2005 [73] and incorporated into JLS version 3 [39], is an indirect criticism of the semantic foundations of concurrency and data, at best.

In general, given the lack of precision about scheduling, little control over thread behavior, and no control over lock fairness, no support for detecting or reasoning about safety and liveness issues, few concurrency researchers dared to stick their noses into the world of Java.

2.2 Modern Java's Concurrency

In 2004, Java 1.5 (aka Java5) was released. It included two major advancements relevant to concurrency:

- JSR 166's "concurrency utilities"—the Java package `java.util.concurrent` and its subpackages—developed by Doug Lea, or what we will generically call *the (Java) concurrency package* henceforth, and
- an extension of the original work by Manson, Pugh, and Adve on the Java Memory Model (JMM). The standardized (via JLS3) JMM provided Java developers and Java Virtual Machine architects with a concrete meaning for what was mandatory and permittable with regards to data synchronization, instruction reordering, and causality within and between threads.

2.2.1 The Concurrency Package

JSR 166's focus was a concurrency API that simplifies the design and development of concurrent Java programs. It does so by providing a libraries of higher-level constructs than the primitive concepts found in earlier versions of Java.

In particular, the concurrency package includes first-class support via API classes for the standard concurrency constructs *atomic variables*, *barriers*, *condition variables*, *explicit locks*, *executors* and *tasks*, *futures*, *latches*, *queues*, *thread pools*, *thread schedulers*, *threadsafe*, *thread-aware*, and *thread-notification collection classes*, *timers*, and *semaphores*.

Java 7 added three new concepts to the concurrency API: support for *fork/join pools*, a helper class for generating pseudo-random numbers without thread contention, and a new synchronization barrier called a *phaser*.

Java 7 also introduced a refinment of multithreaded class loaders to avoid subtle deadlock problems that existed in previous releases.

2.2.2 Other API Changes

Outside of the concurrency package, Java's primitive concurrency support also evolved. In particular, the APIs of the core classes relevant to concurrency (`Object`, `Thread`, `ThreadGroup`, and `ThreadLocal`) grew and changed in small ways.

2.2.3 The Java Memory Model

Prior to 2005, the Java Memory Model (JMM) was under-specified and essentially not understood by all but VM JIT authors. Because of this under-specification, rare programs compiled with different JIT compilers behaved in subtle different ways.

Manson, Pugh, and Adve published a paper in POPL in 2005 that formally characterized the JMM in detail [73]. This formal model was meant to specify the legal behaviors of multithreaded programs and thereby determine the legal optimizations and behavior of VMs and compilers.

This model was adopted by Sun Microsystems for Java 5.0 and a informal description of it is included in JLS3.

All subsequent research and development in Java concurrency is based upon this model, including a large amount of work on data race detection [1, 31], atomicity [30, 34], reasoning about program correctness [23], theories of memory models [13], scheduling [3], model-checking [53, 54], transactions [41], software transactional memory [75], language-based security [10], and more.

2.2.4 Thread Scheduling

Threads come in two variants: *daemon* threads and *non-daemon* (or what we will call *user*) threads. Only *daemon* threads can create *daemon* threads.

The thread that calls the `main` method of a class passed to a newly created VM is a *user* thread. A VM continues to run until either: (1) the `Runtime.exit` method is called, or (2) all *user* threads have stopped running.

Each thread has a priority, encoded as an integer ranging from `Thread.MIN_PRIORITY` to `Thread.MAX_PRIORITY`. Threads whose priority is not specified have a default priority of `Thread.NORM_PRIORITY`. (These values are 1, 10, and 5, respectively, as of Java 7. Priorities are inherited from parent to child threads.

The `java.lang.Thread` class has always included documentation indicating that Java thread scheduling priority-based. It states "Threads with higher priority are executed in preference to threads of lower priority." Unfortunately, this documentation is only indicative, not prescriptive, as many VM releases from Sun and others has not exhibited this behavior reliably.

Some programmers have attempted to tune application performance, avoid bad behavior, or obtain lower UI or I/O latency by playing with thread priorities or using the `Thread.yield` method. None of these techniques have been shown to be reliable, indicating a fundamental failure in the radical underspecification of thread scheduling since the earliest days of Java.

Each thread is in one of six states at any time: *new*, *runnable*, *blocked*, *waiting*, *timed waiting*, and *terminated*. These states are VM thread states, not operating system thread states—the states of the two abstractions need not correlate.

One of the only non-core methods that was not eliminated from Java's `Thread` API in Java's early years is `Thread.interrupt`. Unfortunately, several bugs manifested themselves over the years in this method, particularly when a thread was performing I/O when interrupted on some operating systems. While doing so could leave a stream in an inconsistent state, some programs

rely upon this behavior. Java 7 removed this as the default behavior on the Solaris platform, though it is still available via a VM switch.

2.2.5 Garbage Collection

Java 6 introduced a concurrent mark sweep (CMS) garbage collector (GC). Java 7 further refined this CMS GC to take advantage of faster platforms that were introduced after Java 6's release.

The Real-time Java garbage collector, on the other hand, provides very different guarantees, and exhibits different behavior, than the standard HotSpot GCs.

Fortunately, while there are properties one might wish to reason about relating to the safety and progress of concurrent and parallel programs which are impacted by GC behavior, such is not the subject of this work.

In particular, the formal models described in later chapters are written in a GC-neutral fashion, as they do not have specifications mentioning resource or performance guarantees. Their focus is entirely on functional correctness which includes avoidance of systemic failures like deadlock.

2.3 Alternative Concurrency APIs

Several alternative concurrency APIs have been introduced over the years. While all have not gained sufficient mind-share to see much use in practice, each has influenced the design of the concurrency API, or continues to be useful in its own right.

The main three APIs relevant to this work are:

Oaks and Wong's APIs. In their text *Java Threads* Scott Oaks and Henry Wong introduce an API used to facilitate programming with Java threads using higher-level abstractions such as those found in the mainstream Java concurrency API.

JCSP. Peter Austin, Peter Welch, and Neil Brown from the University of Kent at Canterbury designed and implemented the JCSP library around 1997, far earlier than any other high-level concurrency library was available. JCSP is a directly realization of the CSP model of concurrent and parallel program design and development into a Java API. Classes are directly mapped to concepts such as processes, channels, messages, parallel and choice constructs, etc. Concurrent processes either run within a single VM or across multiple VMs, even on different systems.

CTJ. Developed by Gerald Hilderink and Andy Bakkers at the University of Twente, CTJ also provided a CSP model for Java, much like JCSP.

As none of these APIs have even a fraction of a percent of representation in our case study discussed in chapter 5, we have not used any resources to specify them. We strongly believe that there is utility in creating a native JML model for JCSP, given its quality and the availability of automated tools for reasoning about CSP specification (e.g., FDR [84]).

2.4 Concurrency Development Support

While Java included threads as first-class entities from day one, surprisingly, the Java Developer Kit has never provided any tools for validating or reasoning about concurrent Java programs. In general, there is little support for detecting deadlocks (some VMs perform deadlock detection

and then simply halt the VM) and no support for detecting livelocks, fairness issues, data race conditions, etc.

All in all, this means that developer's have very little tool support for concurrency. They cannot specify a concurrent systems scheduling, they cannot specify, check, or reason about safety or liveness properties, and consequently, they have little control over their concurrent programs.[1]

2.5 Concurrency in Real-time Java

Real-time Java (RTJ) is a different beast than standard Java, especially when it comes to its concurrency and memory models.

The Real-time Specification for Java was proposed and approved as the first ever Java Specification Request, JSR 1 [11]. Its main purpose was to define a variant of Java that focused on soft real-time systems. To accomplish this goal, the Java language itself, its core API, and its concurrency and memory models were modified, much like as was done later for various other Java variants such as JavaCard, Personal Java, etc.

Weaknesses in "classic" Java's scheduling and priority models was a main focus of JSR 1 (see pages 52, 67, and 68 of the RTJ specification). RTJ introduced a variety of changes to accomplish its guiding principles relating to concurrency: predictable execution as a first priority; flexible—but well specified—thread scheduling and dispatching models; priority inversion prevention among real-time Java threads that share a serialized resource; asynchronous event handling, transfer of control, and thread termination.

The first and last mentioned features, in particular, highlights how language features removed from traditional Java were added back into RTJ. In particular, the previously promised well-defined manner in which thread priorities influence scheduling returns, as does the ability to stop a thread's execution, albeit in a safe fashion.

RTJ's API is specified via the `javax.realtime` package. It contains three interfaces, fifteen exceptions, four errors, and fourty-seven classes. A small subset of these classes focus on concurrency, the core of which are two children of `java.lang.Thread`, `javax.realtime.RealtimeThread` and `javax.realtime.NoHeapRealtimeThread`, and the scheduling-related classes, `javax.realtime.Scheduler` and its suppliers. It is these classes that are the focus of our formal model for reasoning about RTJ concurrency.

2.5.1 RTSJ's Memory Model

Real-time Java also has a modified memory model, permitting a much finer degree of control over memory's lifecycle, including allocation, location, and garbage collection (or lack thereof).

As RTJ's memory model predates that of "modern" Java's memory model (JMM) derived from Manson et al.'s work around 2005. Consequently, the subtle changes made to the JMM to (i) accommodate JIT optimizations to increase performance and (ii) reduce ambiguity in update semantics to ensure more consistent behavior across VMs are not present in RTJ.

Due to this fact, as well as the underspecification of scheduling behavior in normal Java, JSR 166's concurrency library is not guaranteed to run correctly on an RTJ VM. And, while RTJ is meant to be completely backwards compatible with normal Java, the authors of JSR 166 suggest that some work is necessary to ensure that the JSR 166 implementation behaves properly in the presence of real-time threads.

[1] Of course, trivial safety properties can be occasionally specified using standard in-line Java assertion statements—a poor man's substitute for a real specification and reasoning framework at best.

2.6 Concurrency in Safety Critical Java

While our focus in on real-time Java, a second variant of Java which focuses on safety-critical systems deserves some mention.

In particular, Safety Critical Java (SCJ), whose early draft specification created via JSR 302, was released in January 2011, five years after the formation of the working group [57]. Its goal is to extend RTJ, as specified in JSR 1, with the minimal set of features necessary for safety critical systems capable of certification, a la DO-178B, level A [86].

While SCJ is not the focus of this deliverable, we have kept its draft specification in mind while doing our work so as to not preclude applying our theory and tools to SCJ in the future.

Chapter 3

Specifications for Concurrency

To reason about concurrent systems written in Java, unless one is only reasoning about basic correctness properties like data races, deadlock, or fairness, one needs to be able to write specifications.

Specifications for Java, in the general, come either in the form of Java annotations, since the release of Java 5, or via pragmas written in specially formatted comments.

Annotations of the former kind are defined either with a particular technology, like those included with the PMD static checker, or are defined as part of several JSRs currently under development, like JSR 305 and JSR 308 [55, 56].

Pragmas of the latter kind are written either like Javadoc tags, as is the case with the commercial tool Jcontract from Parasoft [81], the Contracts for Java tool from Google [20] (which is, in turn, based upon Modern Jass, discussed below), or within specially formatted comments, as is the case with the de facto standard Java Modeling Language (JML) [65].

Jass is an example of a tool that has passed through both of these options. In early versions, specifications were written as if they were Javadoc tags. Jass version 3, which is under development, uses JML is used. Modern Jass, a variant of Jass uses Java annotations to write specifications.

Likewise, JML has several syntactic variants for writing specifications. In the Common JML tools (also known as JML2), specifications are written in special comment blocks, either independent from Javadoc comments, or embedded within Javadoc comments via special XML tags.

Two variations on using Java annotations to write JML specifications have also been developed. In one version, much like in Modern Jass, assertions are written within Java annotations as embedded strings [14]. In the other version, called Java Contracts, assertions are written using nested annotations [68].

While "mainstream" JML focuses on specifying single-threaded modules, extensions to JML that focus on currency represent the richest vein of work to mine.

3.1 Other Static Analysis Tools' Support for Concurrency

Before diving into the details of JML's support for specifying the behavior of concurrent modules, a brief review of other tools' support for such is in order.

The first three tools discussed, CheckStyle, PMD, and FindBugs, do not permit a developer to explicitly specify anything about the functional behavior of a module. The only configurability possible with these tools is the enabling/disabling and tuning of checks, usually both globally and locally [71].

3.1.1 CheckStyle

The lightweight, syntactic, source-level static analysis tool CheckStyle has a only one concurrency-related check: that of the double-checked locking anti-pattern [16, 12].

3.1.2 PMD

The middleweight, syntactic, source-level static analysis tool PMD has several annotations relating to concurrency [82].

DoNotUseThreads. The J2EE specification explicitly forbids the use of threads. Thus, if this check is enabled, any use of `Threads` or `Runnables` raises an error.

AvoidThreadGroup. Surprisingly, the `ThreadGroup` class contains methods that are not thread safe, so it should be avoided by the non-expert concurrent systems developer.

DontCallThreadRun. Explicitly calling `Thread.run()` will execute in the caller's thread of control. This is usually not the intended behavior, and is certainly not the intended use, of this method. `Thread.start()` should be used instead.

UseNotifyAllInsteadOfNotify. Notifying only a single thread monitoring an object moves only one thread in the wait queue on a monitor to the runnable queue, when in fact all threads waiting should be runnable. Thus, it is a best practice to call `Thread.notifyAll()` instead of `Thread.notify()`.

NonThreadSafeSingleton. Incorrect initialization of a singleton object leads to nondeterministic erroneous behavior.

UnsynchronizedStaticDateFormatter. The `SimpleDateFormat` class is not threadsafe, so is often misused.

3.1.3 FindBugs

Likewise, the middleweight, syntactic, bytecode-level static analysis tool FindBugs also has many relevant annotations [27, 6].

In particular, it has over forty checks classified in a *multithreaded correctness* category alone. Most encode semi-standard concurrency patterns and anti-patterns, such as avoiding double-check locking, incorrect synchronization on boxed primitives, misue of the concurrency API, inconsistent design for synchronization or thread safety, possible data race conditions, misuse of the `Thread` class, and more.

Another dozen or so checks that relate to concurrency issues, but which are not strictly classified as correctness errors, are also included. Some of these checks include the following:

FI_EXPLICIT_INVOCATION. `finalize()` methods should not be called explicitely, as this can introduce subtle race conditions, as the VM is responsible for calling this method.

SW_SWING_METHODS_INVOKED_IN_SWING_THREAD. Calling one of several Swing methods from a user thread can result in deadlock.

DMI_FUTILE_ATTEMPT_TO_CHANGE_MAXPOOL_SIZE_OF_SCHEDULED_THREAD_POOL_EXECUTOR. The class `ScheduledThreadPoolExecutor` is not a behavioral subtype of `ThreadPoolExecutor`. In particular, some of its methods do nothing and should not be called. Calling them indicates a potential design flaw.

DMI_SCHEDULED_THREAD_POOL_EXECUTOR_WITH_ZERO_CORE_THREADS. Using a `ScheduledThreadPoolExecutor` with zero core threads will never execute anything, thus represents a design flaw.

SIC_THREADLOCAL_DEADLY_EMBRACE. A subtle incorrect interaction between an instance of a (non-static) inner class and a thread local in an outer class can prevent garbage collection.

STI_INTERRUPTED_ON_CURRENTTHREAD. Calls to `Thread.interrupted()` should be performed statically, not via an unnecessary call to `Thread.currentThread()`.

STI_INTERRUPTED_ON_UNKNOWNTHREAD. Calling `Thread.interrupted()` on a thread object different than the current thread is indicative of an implementation error, as the method is static.

DM_USELESS_THREAD. Creating a thread without specifying a run method, either by deriving from the `Thread` class, or by passing a `Runnable` object, creates a thread with no behavior, and thus just wastes cycles.

3.2 JML Support for Concurrency

The Java Modeling Language is defined by a set of features clustered into "language levels" [67] (Section "Language Levels"). E.g., JML language level 0 are the core features of the specification language that must be implemented by all tools claiming to support JML.

JML language level 'C' are those language features relating to concurrency. Language features in JML 'C', while not mainstream, are supported by several tools such as the Common JML Tools [64], ESC/Java2 [21], and Bandera [45, 22].

Because JML's dominant focus has always been on specifying and supporting reasoning about single-threaded Java programs, work on concurrent constructs is thin. For example, JML does not allow the specification of elaborate temporal properties (except through experimental variants such as those by Huisman and others [89]) nor does it support any modern logical framework or concepts for reasoning about concurrency (e.g., permissions, separation logic, etc.).

What it does support are core concurrency notions relating to Java's most primitive concurrency concepts: object-based locks (monitors), non-termination (divergent behavior), and guards on method invocation. Some of the features are from ESC/Java [35] and others are from Rodriguez et al. [83].

The following section summarizes JML language level 'C'. The purpose of this summary is to provide a foundation for the argument that later work on model-based concurrency specifications (chapter 6) and on higher-order models for concurrency chapter 7 must cover at least these specification language features to be considered complete.

The subsequent section focuses on concurrency annotations relevant to model checking proposed by Rodriguez et al. [83], and the final section of this chapter focuses on annotations introduced by Araujo, Briand, and Labiche for runtime assertion checking of concurrent programs [4].

It should be noted that Flanagan and Freund have published a large amount of work in the verification of concurrent Java programs (e.g., [34]). While their work has influenced the design of JML language level 'C' (henceforth written JML_C) as well as the aforementioned research, because they do not typically support JML, and because they do not ship the tools that they build and with which they perform experiments, we choose not to discuss their work at length here.

3.3 Core Concurrency Constructs

The core concurrency constructs in JML_C fall into three clusters: (i) *commit points* and *when* clauses, (ii) *monitored* and *monitors for* clauses, and (iii) *locks/locksets* and their operators. The first cluster focuses upon

3.3.1 Commit Points and When clauses

When programming concurrent systems, it is often necessary to specify what must be true before and after a block of code is executed. One can specify such properties via assertions, but when an assertion fails the program halts. The kind of properties that are useful in a concurrent setting are more akin to *guards* and *transactions*.

A *guard* on a block of code stipulates a predicate that must hold true before the code can execute. Guards are specified in JML using the `when` clause, originally proposed by Lerner [69].

A `when` clause (specified via the non-terminal ⟨*when-clause*⟩) is of the form:

⟨*when-clause*⟩ ::= '`when`' ⟨*predicate*⟩ '`;`'

The meaning of this specification is that a calling method will be delayed until the condition given in the `when` clause holds, presumably by a concurrent thread *and* will not complete execution (see the following paragraph) until the `when` clause holds. The specification says nothing about how the guard will become true, only that it must become true.

To check a *when* clause, the complementary *commit* clause is used. A *commit point* is a statement label in the code. If there is no explicit commit label, then the implicit commit point is the exit point of the method. At the commit point the property specified in a method's *when* clause must hold. There is no guarantee that the method will proceed the first time this condition holds, so the condition may have to hold many times before the thread may proceed to its commit point.

3.3.2 Monitored and Monitors-for clauses

The `monitored` modifier is used on a non-model field declaration. Its meaning is that a thread must hold a lock on the object that contains the field (i.e., the `this` object containing the field in question) because it may read *or* write to the field [35]. An example use of the `monitored` modifier is found in Listing 3.1.

Complementing the `monitored` keyword is the `monitors_for` clause, adapted from ESC/-Java [35]. This clause has the following basic syntax:

⟨*monitors-for-clause*⟩ ::= '`monitors_for`' ⟨*identifier*⟩ ('`=`' | '`<-`') ⟨*spec-expression-list*⟩ '`;`'

A `monitors_for` clauses, such as seen in lines 4–6 of Listing 3.1, specifies a relationship between a field (in this case, the fields `c`, `s`, and `d`) and a set of objects (`this`, `MonitoredExample.class`, and `this` and `o`, respectively).[1]

The meaning of these declarations is that *all* of the (non-null) object in the list must be locked to access (read *or* write) the related field. Note that there is no proscription on the order in which multiple locks must be obtained, as that is specified using other constructs discussed below.

[1] The set of objects is actually a specification expression list, so need not be literals as in this simple example, but instead, e.g., the result of a call to a pure method that returns a reference to an object.

```
1  public class MonitoredExample {
2    /*@ monitored @*/ int i, Object o;
3    float f;
4    char c;   //@ monitors_for this;
5    short s;  //@ monitors_for MonitoredExample.class;
6    double d; //@ monitors_for this, o;
7    public void n(int g) {
8      i = g; // illegal access due to lack of lock acquisition
9      f = g; // legal no matter what
10   }
11   public synchronized void o(int g) {
12     i = g; // legal access due to lock acquisition
13     f = g; // legal no matter what
14   }
15   public static void p(int g) {
16     i = g; // illegal access due to lack of lock acquisition
17     f = g; // legal no matter what
18   }
19   public synchronized static void p(int g) {
20     i = g; // illegal access due to lack of proper lock acquisition
21            // (lock acquired is on class, not on object)
22     f = g; // legal no matter what
23   }
24 }
25
26 class ClientExample {
27   void m() {
28     MonitoredExample me = new MonitoredExample();
29     me.i = 1; // illegal use
30     Object p = me.o; // illegal use
31     me.f = 0; // legal use
32     synchronized(me) {
33       me.i = 2; // legal access due to lock acquisition
34       p = me.o; // legal access due to lock acquisition
35       me.f = 1; // legal access
36     }
37     Object q = me;
38     synchronized(q) {
39       me.i = 3; // legal access due to aliased lock acquisition
40     }
```

Listing 3.1: Monitored and Monitors For Examples.

3.3.3 Locksets

In order to specify the order in which a set of locks must be locked to access a resource protected by multiple locks, several specification constructs are used in JML$_C$.

3.3.3.1 Specifying Lock Ordering

Firstly, one must be able to denote the order in which locks must be acquired. Order is specified in JML$_C$ using the < and <= binary infix operators which operate on objects, as any object can be used as a lock. Lock orders are specified via JML's `axiom` clauses, thus are statically specified.

For example, consider the following Java program fragment:

```
Object o, p, q;
//@ axiom o < p & p <= q;
```

In this program, the object referenced via the variables `o`, `p`, and `q`, if they are used as locks, must be acquired in a particular order: `o` must be acquired first and then either `p` or `q` may be acquired. In this example, lock `p` is *larger than* lock `o`.

Each axiomatic specification of lock ordering induces a partial order on locks. The complete set of such specification for a program must, in turn, be a partial order. If there is a problem with such a specification, e.g., a cycle exists in lock relations, then tools supporting lock orders (such as ESC/Java2) report an error.

These operators can also be used to test the order of locks within a specification expression, though this use is rarely seen in practice. When used in this way, the sub-expressions to either side of the operator must be reference types, and the result is of type boolean.

3.3.3.2 Locksets

Secondly, a *lockset expression* is used to specify the (dynamic) set of locks that the current thread holds. Its syntax is simply `\lockset()` [35]. The type of this expression is `JMLObjectSet`, i.e., a set of references to arbitrary objects.

3.3.3.3 Maximal Locks

Finally, a *max expression* is used to identify the "largest" lock in a set of lock objects, in the sense of the lock ordering denotation of a program.

The syntax of a *max expression is as follows*:

⟨*max-expression*⟩ ::= '`\max`' '`(`' ⟨*spec-expression*⟩ '`)`'

Given the above example, the value of `max(@ o, p, q @)` is either `p` or `q`, as neither is larger than the other given the specification.[2]

[2]The use of '`@{`' and its sister are inline set comprehensions, a little-known-but-useful syntactic construct available in JML.

Chapter 4

Existing Tool Support for Reasoning about Concurrency

While developing correct concurrent and parallel applications has been, and will continue to be, a challenge, surprising few tools exist to support such. In fact, with the rise in multi-core processors, one would think that there would be a corresponding rise in the number and quality of commercial and open source tools to support validation and verification of concurrent software, but this is not the case.

This chapter surveys the state of existing, publicly available commercial and free tools so as to contextualize the main contributions of this work discussed in later chapters, that of model-based specifications for concurrency and parallelism, reasoning about such specification through multiple refinement levels, and the foundations for verification of multi-threaded clients via native higher-order types.

4.1 Built-in Support in the Java Developer's Kit

In general, there is little built-in support for reasoning about or validating multi-threaded Java software. The Java Language Specification and the Java Virtual Machine Specification (all versions) contain no mandate for support for standard best-practices like runtime deadlock detection, fairness guarantees, nor race condition detection [5, 37, 38, 39, 40, 70].

> The Java programming language neither prevents nor requires detection of deadlock conditions. Programs where threads hold (directly or indirectly) locks on multiple objects should use conventional techniques for deadlock avoidance, creating higher-level locking primitives that do not deadlock, if necessary. (JLS, Java SE 7 Edition, 2012-02-06, page 564) [40]

Some Java virtual machines *do* support deadlock detection. In general, when a VM detects a cycle in lock dependencies, it halts the entire VM and reports a stack trace for all threads.

4.2 Commercial Tools

Nearly a dozen commercial tools with tacit or explicit support for multi-threaded systems have come and gone over the past fifteen years. Some of these tools include:

- Allinea's DDT debugger which focuses on C, C++, Fortran, and CUDA analysis on several workstations, supercomputers, processors and compilers,
- the DiViA framework consisting of the ExMon execution monitor, the MPD multi-process debugger, the BAND automatic bottleneck detector, and other tools not relevant to understanding multi-threaded applications, all of which target C programs,
- Intel's tool suite, Intel Parallel Studio, for debugging multi-threaded programs, which also focus on C/C++ or Fortran programs using OpenMP,
- the JProbe tool from Quest Software, which while primarily is a high-end profiling tool for Java, includes support for runtime analysis of the behavior of threads, including deadlock and livelock/starvation detection and data-race detection,
- the JProfiler tool, which also focuses on Java, from ej-technologies, which includes runtime analysis of thread behavior, but no deeper analysis like that which JProbe offers,
- Berkeley's Mantis Parallel Debugger, which focuses on the Split-C concurrent programming language and ran on Thinking Machine's CM-5 supercomputers,
- IBM's Parallel Environment, which includes their parallel debugger (PDB) and their High Performance Computing Toolkit, all of which focus on C/C++, and Fortran programs running on IBM hardware,
- The Portland Group's PGI graphical symbolic debugger, which focuses on MPI and OpenMP applications written in C/C++ and Fortran code,
- Rogue Wave Software's TotalView debugger, which, again, focuses on C/C++ and Fortran code that uses threads, OpenMP, MPI, or GPUs, and
- YourKit's Java Profiler, which includes thread telemetry information, automatic deadlock detection, and monitor analysis like JProbe and JProfiler.

While three of these commercial tools do support Java program analysis and debugging, none permit one to reason about code statically, nor do any of them publish information about the means by which they perform runtime verification. Consequently, it is unknown if their runtime analysis is sound.

Also, none of these tools permit the specification of properties relevant to concurrent system correctness. In particular, none permit a developer to specify the static or dynamic concurrency properties of a system architecture or design using high-level models like UML or other model-based specification languages. All have a basic built-in notion of what constitutes a well-behaved multi-threaded system, and that notion is neither parameterized or user-configurable.

4.3 Non-commercial Unavailable Research Tools

There are also a number of research tools that have been developed to support reasoning about multi-threaded programs. Some have turned into commercially supported tools, for example the aforementioned MPD debugger. As the size of the set of such tools is quite large, we only focus on those that relate to reasoning about or monitoring Java programs.

The bulk of software produced to dynamically and statically analyze multi-threaded Java programs come from Flanagan, Freund, and their collaborators [34, 28, 30, 1, 31, 29, 32, 33]. In the literature they discuss over half a dozen tools developed over the past decade to perform race condition analysis, atomicity analysis, thread-modular reasoning, interference analysis, and more. Unfortunately, all but one of these tools have never been released to the public.

There are also a few tools that augment the behavior of Java VMs to perform runtime monitoring or deterministic debugging (e.g., DejaVu for the Jalapeno JVM, ODR from Berkeley, and

a number of tools from Gopalakrishnan's group in Utah), but none permit one to reason about models of concurrent software.

4.4 Open Source Tools

Finally, there are a handful of open source tools that are publicly available, which, surprisingly, nearly all come from industry with academic involvement. They include:

- Microsoft Research's Alpaca and CHESS tools, which focus on finding and reproducing bugs in concurrent .Net programs via automatic, deterministic runtime verification [85, 78],
- ESC/Java2, which includes support for statically reasoning about thread correctness conditions like deadlock [35],
- Hatcliff et al.'s Bandara model checker for Java [45],
- Araujo et al.'s parallel runtime assertion checker [4], and
- Beckert and Klebanov's concurrency variant of the KeY system [7].

Many of these tools have utility, and moreover most are part of our standard collection of tools for concurrent and parallel systems' design and development. But once again, none permit one to specify or reason about concurrent and parallel software at the model level. The closest approximation are the concurrency-related JML annotations, discussed in section 3.2. Each Java-related tool supports a subset of these annotation, but all exist solely at the source level.

Chapter 5

Analyzing the Use of Concurrency APIs

To better understand how one should specify concurrent code, where one should focus resource-limited activities like specification writing, and how to prioritize native models mechanization, we felt it necessary to perform a case study to answer the question, "What concurrency constructs do real Java developers *actually* use?"

5.1 Empirical Evidence of Concurrency Use

This case study is a quantitative analysis of a very large corpus (roughly half a millon Java classes in size, written by thousands of developers) of production-quality, multi-threaded Java code at the bytecode level.

By looking at the frequency and patterns of use of a variety of concurrency constructs, both low- and high-level, we can characterize developer behavior and prioritize specification writing, heuristic focus, and reasoning goals.

To effect this large-scale analysis, a new tool called the CHARTER Histogram tool (or just the Histogram tool, for short), was developed.

5.2 The Histogram Tool

The Histogram tool [9] analyzes units of byte code given on the command line. A unit of byte code is either a ZIP archive, such as `.jar`, `.war` and `.zip` files, or a directory. In each case, the unit is scanned for Java classfiles, and all the byte code in those classfiles is consequently scanned. The tool builds two histograms: one contains the number of occurrences of classes and the other the occurrences of method invocatoins. The output is written in CSV format.

5.3 Concurrency Use Code Corpus

The main dataset used is the Qualitas Corpus, Version 20101126r from February 2009, from the Qualitas Research Group at the University of Auckland [87]. This data set consists of 105 system, totalling 1.2GB of jar files containing roughly half a million classes. Nearly all of the systems (97) use the `Thread` class. Roughly two-thirds of the systems (66) make use of the `java.util.concurrency` framework. Other concurrency libraries were used in only 3 projects.

We also collected the distribution archives of every project produced by the Oracle/Sun, Apache Foundation, the Eclipse Foundation, and others. A cursory look at this even larger (but not disjoint) dataset confirms the concrete analysis we performed on the Qualitas Corpus.

5.4 Analysis Results

With the histogram tool, we analyzed the numbers of occurrences of classes and methods from the `java.util.concurrency` framework in the bytecode of the Qualitas Corpus data set. In this report, we include the top 25 classes and methods used, measured in the number of references and measured in the number project using them. See Tables A.1, A.2, A.3, and A.4 in Appendix A.

5.4.1 Clustering of Concepts

Clustering this analysis based upon framework design and developer usage patterns, we see six kinds of concurrency concepts:

- a variety of locks, now expressed as first-order classes: `Lock`, `ReentrantLock`, `ReentrantReadWriteLock`, `ReadWriteLock`, and `Condition`,
- a variety of atomic variables: `AtomicInteger`, `AtomicLong`, and `AtomicBoolean`,
- the executor framework: `Executors`, `Executor`, `ThreadPoolExecutor`, `ExecutorService`, `Future`, `FutureTask`, `ScheduledExecutorService`, and `TimeUnit`,
- thread-safe maps: `ConcurrentMap` and `ConcurrentHashMap`,
- thread-safe queues: `BlockingQueue`, `ConcurrentLinkedQueue`, `LinkedBlockingQueue`, and
- miscellaneous classes, like `CopyOnWriteArrayList`, `CountDownLatch`, and `Semaphore`.

5.4.2 Prioritization Results

Given the results of the static analysis, it is clear that: (i) at least one class from each cluster must be specified, so as to begin to formulate a specification template or pattern for said cluster, (ii) the choice of which class to specify in a cluster is often obvious (e.g., in the case of `Lock`, `ConcurrentHashMap`, and `AtomicInteger`).

It makes no sense to specify or analyze some classes in isolation because they are part of a larger framework (e.g., the executor framework).

Finally, some classes do not deserve early specification due to developer use, but may due to foundational meaning (e.g., `Semaphore`).

5.4.3 Analysis Surprises

There are a small handful of surprises that arise out of this objective, quantatative analysis.

Firstly, the `CountDownLatch` is far, far more popular than `CyclicBarrier`, even though the latter has a simpler semantics and is, in our experience, typically taught much more frequently in concurrency courses.

Archetypical constructs available at the language level, like semaphores, various kind of queues, and futures, occur with very low frequencies. We have no working hypothesis for why this is the case beyond the suspicion that, as instruction in concurrency theory has fallen out of favor in universities, so too has appreciation for these basic constructs.

Finally, atomic *primitive* type wrappers are used enormously more often than atomic *reference* wrappers. We suspect this is due to the fact that developers rightly believe that reference updates are atomic, but forget that a test-and-set on a reference is not.

Chapter 6

Model-based, Refinement-centric Concurrency Annotations

Concurrency is no longer a non-functional property of a system, as most desktop systems and even some mobile devices are now multicore or multithreaded. Consequently, the concurrency properties of a system should not be an afterthought. Unfortunately, while there are a few basic concurrency design patterns, they are not commonly used in model-driven development and they lack formal semantics. In this work we introduce a general-purpose family of concurrency annotations with a formal semantics. We apply these annotations to high-level models like those expressed in UML, medium-level designs like those expressed in OCL, and low-level development artifacts like inline documentation and program code. We also define a reversible refinement relation for these annotations among three languages: the EBON system specification language, the Java Modeling Language, and the Java programming language. These annotations can be statically and dynamically checked with respect to refinement, system design, runtime validation, and static verification.

6.1 Introduction

Concurrency properties rarely appear in architecture descriptions beyond a vague mention of "thread-safe" and "scalable." The UML superstructure specification [] and two UML profiles [] (but not the infrastructure specification []) do briefly mention concurrency. However, they provide only a handful of weakly specified concurrency-related constructs, most of which are relegated to narrow domains like realtime and embedded systems.

Some lower-level annotation languages do include some mention of concurrency, though OCL [91], the constraint language used with UML specifications, is not one of them. For example, the Java Modeling Language (JML), the *de facto* standard specification language for Java, includes a number of concurrency-related annotations that we discuss in detail later.

Finally, modern programming languages have numerous built-in and external notions of concurrency, primarily in the form of first-class threads, processes, locks, schedulers, and more.

Our thesis is that modern system design languages would be well-served by the introduction of (1) first-class concurrency modeling primitives that have a formal semantics, (2) formally defined safety-preserving refinement relations among development artifacts like requirements, designs, annotations, and programs, and (3) tool support for authoring, generating, reasoning about, validating and verifying concurrency specifications. We introduce a concurrency modeling primitive called the *concurrency semantic property*, an instance of a *semantic property* [60],

that encompasses all three of these features.

These concurrency annotations are primarily meant to be *manually* written by modelers or developers at the model level about applications, libraries, frameworks, or data structures. The annotations are then *automatically* refined into detailed specifications at the source level, relieving a significant amount of specification burden. In particular, because the concurrency semantic property is essentially a domain-specific language of concurrency, it is very expressive. As is typical with model-driven techniques, one or two lines of annotation at the model level can be transformed into dozens of lines of specification at the source level. Moreover, it is significantly easier to write and reason about concurrency at the model level. Consequently, since our refinement is meant to guarantee safety and progress preservation, we hope to stop reasoning about ad hoc concurrency specifications at the source level.

These primitives were originally derived from work in parallel programming archetypes in the mid-1990s [19] and incorporated into a language-independent code standard in 1996 [61]. As such, they have been used to specify the concurrency properties of, in aggregate, tens of thousands of lines of Java and Eiffel code over nearly the past 15 years. Only recently, as we have begun to reason about concurrent Java systems specified in JML, has there been a need to provide a formal semantics.

In this work, we use three languages that constitute four refinement levels. The highest level is object-oriented models written in the high-level modeling language EBON (the Extended Business Object Notation [58]), an extensible architecture description language whose ancestor, BON, was originally designed for use with Eiffel [90]. The next level is Java class skeletons with concurrency properties encoded in source-level annotations and documentation and other behavioral specifications written in the aforementioned JML [66], a model-based behavioral interface specification language in the Larch tradition [42]. The third level is the result of mapping the source-level concurrency annotations to more detailed JML specifications, and the fourth level is the final Java program code itself.[1]

Note that these annotations focus exclusively on safety and progress properties, not orthogonal, non-functional properties of performance and scalability. Nascent support for specifying these other properties already exists in JML in the form of the `duration` clause. Lifting such low-level specifications into semantic properties at the model level is an exercise that complements this work.

For the remainder of this paper, we will focus primarily on the high-level concurrency properties in EBON and their mapping to detailed JML specifications, and will briefly discuss the intermediate refinement of concurrency properties to annotations and documentation. The classic sleeping barber concurrency problem is used to illustrate the use of these annotations and their refinement. Other examples are found on the accompanying website.[2]

The following subsections introduce EBON, JML and semantic properties in turn. The reader who is already familiar with any of these is welcome to skip the corresponding sections.

6.1.1 EBON

The Business Object Notation (BON) was introduced by Walden and Nerson in 1994 for use with the Eiffel language [90]. BON is a high-level modeling language used to describe, in textual or graphical form, the key artifacts associated with the analysis and design of object-oriented

[1] We can also refine concurrency properties to the object code level via annotations and proofs in bytecode, but we will not discuss object code here.

[2] http://kindsoftware.com/products/opensource/Propi/concurrency_examples.html

architectures including concept analysis, static and dynamic models, scenarios and events, and more. BON can be thought of as a "micro-UML with semantics." The entire language is typically taught in two lecture sessions and has both a semi-formal and a formal semantics, the former of which is described in the aforementioned text and realized in the EiffelStudio tool and the latter of which is mechanized in PVS [26].

EBON is a new version of BON introduced by Kiniry in 2002 [58], which enables the BON high-level specification language itself to be extended with new notation and new semantics. We use EBON with the addition of our concurrency semantic property as the high-level modeling language in this paper.

6.1.2 JML

The Java Modeling Language [66] is a behavioral interface specification language for Java. JML is a rich language with dozens of constructs, yet at its core it is a simple formal specification language that lets one naturally specify type contracts via invariants on classes and interfaces and preconditions and postconditions on methods. It is also possible to specify mathematical models using JML that describe (parts of) a system abstractly and to stipulate refinement relations between such abstract models and the concrete system. Additional JML constructs, such as ghost fields, datagroups, and non-null and ownership types, enable the specification of non-trivial systems [18]. JML is supported by a large set of tools that enable validation and verification of JML-annotated software systems [15].

6.1.2.1 Concurrency Support in JML

Recall that in section 3.2 we explained in detail the constructs available in core JML to support specification of concurrency-related concepts. This shorter, more focused debriefing on that same material is summarized here for convenience to the reader in what follows.

JML currently includes the following five concurrency constructs, some of which originated with ESC/Java [35] and were further developed by Rodríguez et al. [83]

The `monitors_for` clause specifies the locks that are used to protect specific data. It always includes one field and one or more references to locks. For example, `monitors_for f <- e1, e2` stipulates a relationship between field `f` and the locks referenced by `e1` and `e2`. The meaning of this declaration is that all of the non-null objects on the right-hand side must be locked to access the field on the left-hand side.

The `when` clause specifies that execution of a method suspends until certain conditions are met. For example, the specification `when P` on a method `m` means that a caller of `m` will be suspended at its *commit point* until predicate `P` holds [69]. The commit point is the point just before a statement labeled `commit`; if no commit point is specified, the implicit commit point is the end of `m` and `m` is not allowed to complete until `P` holds. No guarantee exists that `m` will proceed past its commit point the *first* time `P` becomes true, so `P` may have to become true many times for `m` to proceed. The `when` clause does not provide a mechanism to assign responsibility for establishing `P`; in an implementation, `P` might be established by a scheduler or a transaction system, or might never be established (causing `m` to never terminate).

The `\lockset` expression returns a `JMLObjectSet` (a JML model that denotes a set of Java objects) containing the locks held by the current thread.

The Boolean binary infix operators `<#` and `<#=` are used to specify and reason about lock ordering. These operators denote the order in which locks must be acquired, where "larger" locks must be acquired after "smaller" locks. This ordering is global; if an ordering is specified between

any pair of locks in any invariant of a system, it applies to those locks regardless of where in the system they are used. ESC/Java2 supports reasoning about the well-formedness of lock ordering specifications and other related concurrency properties [59].

Finally, the `\max` expression is used to determine the "largest" lock in a set of locks. It takes a well-ordered lock set (that is, a lock set in which a unique ordering has been specified among the locks using `<#`) as an argument and returns the largest lock in the set.

As currently defined, JML concurrency constructs only consider Java's implicit synchronization locks and child classes of `java.util.concurrent.lock.Lock` to be locks. Moreover, in the existing JML model, a lock has only two states: locked and unlocked. For the purposes of this work, we consider a lock to be a more general concurrency construct that may potentially be held by more than one thread at once; for example, if lock `l` is implemented with a semaphore (and used with proper semaphore discipline), the initial value of the semaphore determines how many threads may simultaneously hold lock `l`. Thus, in our usage, we effectively redefine the existing JML constructs to encompass our broader definition of locks; for instance, more than one thread may have the same lock in its `\lockset` simultaneously.

Within this work we lift all the existing JML concurrency constructs into the EBON language. Thus, there are new specification cases for features (`when`), new expressions for invariants (`monitors_for`), a new keyword to denote the current thread's lockset (`Lockset`), and new operators and expressions for specifying and reasoning about lock ordering (`<#`, `<#=` and `max`, respectively).

Note that adding these constructs to a specification language need not be done by extending the language as we do here. Instead, one can specify a cluster of classes that embody the constructs at the design, rather than language, level. For instance, a class `LOCKSET` could be specified with features such as `max: LOCK`, `size: INTEGER`, and `in: BOOLEAN -> l: LOCK`. The semantics of these operators would then be defined by the contracts of the classes in the new cluster.

6.1.3 Semantic Properties

This work is couched within Kiniry and Fairmichael's *semantic properties* [60]. Semantic properties are domain-specific, refinement-centric specification constructs used to augment existing specification languages. This is exactly our intention, as we wish to augment the EBON, JML, and Java languages with concurrency properties in a uniform fashion. Consequently, we introduce a single new semantic property in this paper, the *concurrency semantic property*, by describing its use, semantics, and refinement.

The remainder of the paper is organized as follows. The next section discusses related work in this area. section 6.3 introduces the *concurrency semantic property* itself and details its meaning and usage. section 6.4 provides a concrete semantics of the concurrency semantic property encoded in JML-annotated Java. Finally, section 6.5 illustrates the use of the concurrency semantic property to specify a solution to a canonical synchronization problem. We conclude with some reflections and observations about this work and the directions in which it can be taken.

6.2 Related Work

The earliest work that focuses on specifying and reasoning about concurrency in a purely object-oriented context is by Lerner [69]. It was from his work that the aforementioned `when` construct originated.

The main thrust of work in specifying and reasoning about concurrent systems in Java is by Rodríguez et al. [83], Corbett et al. [22], Flanagan, Freund, Qadeer, Abadi et al. [2, 35], and Hurlin [43, 46, 47, 48].

Rodríguez et al.'s work was the first to focus seriously on adding and reasoning about new JML concurrency constructs. They use model checking to validate the correctness of concurrent Java programs against JML specifications. They also use an ownership type system to denote aliasing and ownership patterns to avoid unnecessary locking forced by representation exposure.[3] Several other constructs not mentioned above are proposed by this work, including the `locks` clause, the `\lock_protected`, `\thread_local` and `thread_safe` constructs, the module and method modifier `atomic`, the `\independent` predicate, and the `locked_if` clause.

It is unclear how many of these constructs need be lifted to the model level; more case studies are necessary. The only constructs that are ubiquitously used in Rodríguez et al.'s case study are `atomic`, `\thread_safe`, and `\independent`. Only the first two of these have utility at the model level, and we lift them to the model level here as `atomic` and `concurrent` properties, respectively.

Hurlin and his collaborators' work focuses on augmenting JML with separation logic operators to reason about unstructured heaps in the presence of concurrency. While their work provides constructs and verification techniques that are legitimate targets of refinement from our EBON specifications, none directly impact this work as they do not contribute new design level elements. Their protocols can be modeled using EBON dynamic diagrams and their parallelism constructs all map into the concurrency semantic property.

Flanagan and his collaborators' work focuses on atomicity, races, and modular verification, mainly through the use of novel type systems and model checking. As such it complements, rather than competes with, this line of research.

More generic and recent work focusing on reasoning about concurrent algorithms by Jacobs et al. [49, 51, 52], much like the work of Hurlin and Flanagan, seems to naturally lift to the design level using our techniques.

Mizuno's work focuses on a global invariant-based approach to the specification of concurrent algorithms at a high level. This approach has been applied to the Java and C programming languages [76]. Mizuno et al. also mention its use in UML-like languages [77], but do not discuss the means by which one specifies and reasons about high-level concurrency primitives in object-oriented models beyond synchronization regions and global invariants in use-cases.

Some basic tool support for concurrency specification and refinement is available via the *SyncGen* system from KSU [62], described partly in a paper by Deng et al. [24] SyncGen generates executable Java or C implementations of concurrent algorithms from concurrency pattern specifications and template implementation files. Generated implementations in Java are verified using the Java model-checking tool set Bandera [45, 22]. Consequently, this work is similar to ours in that it shares the property of reasoning about lower levels of refinement (i.e., annotated program code) and verifies properties of higher levels (i.e., *SyncGen* specifications). However, it focuses only on the low-level Java concurrency primitives that were available prior to Java 5, concurrency descriptions only exist at the highest level of specification, and the approach is not reversible.

[3] We specify such patterns using EBON's creator charts, but that is not the focus of this paper.

6.3 The Concurrency Semantic Property

The concurrency semantic property allows a developer or system designer to annotate *classes* and *features* (methods and constructors), specifying system behavior with regard to concurrent execution. We define six subtypes of the concurrency semantic property: `concurrent`, `sequential`, `guarded`, `failure`, `atomic` and `special`.

Over the past 15 years we have used (most of) these constructs to specify the concurrency properties of numerous systems. New constructs have been added as new specification challenges arose, and constructs' semantics have evolved in tandem with concurrency architecture styles. Consequently, we believe that this set of six subtypes covers essentially all standard modern concurrency architecture patterns, bar those mentioned in the conclusion that we explicitly do not handle, e.g., lock-free programming or the absence of a locking discipline.

Informally, annotating a class with concurrency semantic properties is the same as annotating every feature contained within that class with the same properties. In this respect it is similar to the use of the JML modifier `pure` on a Java type, which results in all constructors and non-static methods for that type being considered `pure`. Like `pure` on a type, concurrent properties on a class do not restrict the behavior of methods that are newly introduced by a child class. Our semantics differs slightly, however, in that a class-level concurrency property does not apply to inherited methods that the class does not reimplement. A class-level concurrency property is meant to be equivalent to annotating only the features explicitly defined in the class with that property; for example, it would not make sense to annotate the `getClass` method of `java.lang.Object`, which is inherited by every Java class, with a concurrency property that `java.lang.Object` did not implement.

As previously stated, we consider a lock to be a general concurrency construct that may potentially be held by more than one thread simultaneously. This definition is the one we use throughout this section. In addition, all thread counts used in our definitions are *reentrant*; a thread executing a feature recursively is counted as a single thread regardless of the recursion depth.

6.3.1 Concurrent

The `concurrent` property specifies a limit on the number of threads that may safely concurrently execute a feature. Threads are counted on a per-feature, per-object basis.

Definition 1 (Feature Thread Count)**.** *The thread count for a feature `f` of object `o` is the number of threads simultaneously executing `f` on object `o`.*

If the concurrent limit is exceeded, and no `failure` property is specified (see subsection 6.3.5), the object is considered *broken* and its behavior is no longer guaranteed.

Definition 2 (Broken Object)**.** *When an object is broken, the object's invariants and feature postconditions are no longer guaranteed.*

With the ability to break an object, each invariant and postcondition predicate P effectively becomes $\neg broken \Rightarrow P$. Thus, when the *broken* flag is set every invariant and postcondition of the object is trivially true and no guarantees are made about the object's behavior.

Definition 3 (Concurrent)**.** *The declaration `concurrent N` on a feature means that if the thread count on that feature does not exceed `N` for a given object, that object's invariants continue to hold and the feature's postcondition continues to be guaranteed. If the thread count exceeds `N` during execution, the postcondition becomes* true *and the object becomes broken.*

It is erroneous to annotate a class or feature with more than one `concurrent` property, as there can only be one bound beyond which an object becomes broken. Child classes may increase, and must not decrease, the `concurrent` bound of an inherited feature. By default, all features are `concurrent` ∞ unless a bound is specified explicitly or inherited from parent class features.

The `concurrent` property necessitates a per-feature, per-object count of the number of simultaneous threads; we discuss this further in subsection 6.4.1.

6.3.2 Sequential

The `sequential` property specifies that a feature is not thread-safe. No behavior is guaranteed if multiple threads concurrently execute the feature on the same object.

Definition 4 (Sequential)**.** *The declaration `sequential` is equivalent to the declaration `concurrent 1`.*

Because `sequential` is equivalent to `concurrent 1`, it is erroneous to declare both `sequential` and `concurrent` properties on the same class or feature.

6.3.3 Locks

The set of locks that may be acquired during execution of a feature is an important aspect of concurrency behavior. The `locks` property specifies this set of locks. A thread executing a feature may only attempt to acquire the locks specified in the `locks` property. The thread is not compelled to attempt to acquire *all* the specified locks on every execution, although this will often be the case.

Definition 5 (Locks)**.** *A declaration of `locks` L_1, L_2, ..., L_N on feature `f` provides an upper bound on the set of locks a thread may try to acquire during execution of `f`. If the set of locks is `Void`, a thread may not try to acquire any locks during execution of `f`. If the set of locks is `*`, a thread may acquire any set of locks during execution of `f`.*

The use of multiple `locks` properties on a single feature is semantically equivalent to the use of a single `locks` property with the union of all the locks from the multiple `locks` properties. The `locks` property specifies neither lock ordering, which is handled in invariants by using the `<#` and `<#=` operators, nor the exact nature of the named locks. The default `locks` property is `locks Void`.

By default, a reimplemented feature has the same `locks` property as the original feature. To more strictly constrain locking behavior, a child class may use a `locks` property on a reimplemented feature with a subset of the inherited locks. For instance, if a class inherited a feature `f` declared with `locks A, B`, it would be legal for the class to redeclare feature `f` with `locks Void` to indicate that the new implementation is lock-free. However, a class may not relax the constraints on a reimplemented feature's locks by adding locks that were not specified in the original feature declaration. In particular, because the default `locks` property is `locks Void`, classes may not add `locks` properties to inherited features that were not originally declared with `locks` properties.

6.3.4 Guarded

The `guarded` property strictly specifies the locking and blocking behavior of a feature. Each lock that a thread is required to hold before it can proceed to execute the feature should be listed in a `guarded` property. A thread calling the feature may block until it acquires the necessary locks.

A thread may acquire all the locks specified for a given feature before calling it, but this is not required. If a thread does hold all specified locks before calling the feature, it can expect its execution of the feature to proceed without blocking. Each lock specified in `guarded` that was not held by the thread prior to calling the feature must be released before or upon feature termination.

Definition 6 (Guarded)**.** *A declaration of* `guarded` L_1, L_2, ..., L_N *on feature* `f` *specifies the set of locks that a thread must hold before it can proceed to execute* `f`*. Before or upon the termination of* `f`*, the thread must release all specified locks that it did not hold before calling* `f`*.*

All locks specified in a feature's `guarded` property are implicitly added to the feature's `locks` property if they do not already appear there. Like `locks`, `guarded` specifies neither lock ordering nor the exact nature of the named locks.

A reimplemented feature has the same `guarded` property as the original feature, and this property may not be changed in any way. Child classes cannot add new `guarded` locks because that would violate the inheritance rule for `locks`, and cannot remove `guarded` locks because clients may depend on the use of those locks as stated in the original `guarded` property.

6.3.4.1 Guarding on a Feature

For the convenience of developers, every feature of every object has an implicit counting semaphore. This makes it possible to limit the number of concurrent threads executing a particular feature on a per-object basis. The implicit semaphore for a feature `f`, denoted L_f, can only be accessed through the use of the `guarded` property on `f` and is not otherwise exposed.

Definition 7 (Guarded on a Feature)**.** `guarded N` *for positive integer* `N` *on feature* `f` *is equivalent to* `guarded` L_f*, where* L_f *is a counting semaphore with initial value* `N`*.*

For example, declaring `guarded 2` on feature `f` of class `C` means that at most two threads may call feature `f` simultaneously on each instance of `C`. Any additional thread that attempts to call `f` will suspend until there are fewer than two threads executing `f`.

Only one `guarded N` property is allowed per feature, and the limit `N` may not change in a reimplemented feature for the same reason that the list of locks may not change.

Guarding on a feature and on other locks simultaneously is permitted, and it is legal to declare `N` and other named locks in a single `guarded` declaration. In such situations, the calling thread attempts to acquire the implicit feature semaphore immediately upon entry to the feature, regardless of the subset of the other locks the thread already holds. There is no way to specify a lock ordering relationship between the implicit feature semaphore and explicitly-listed locks.

6.3.5 Failure

The `failure` property specifies an exception to be thrown immediately when the `concurrent` bound would otherwise be exceeded.[4] Thus, it must be used in conjunction with a `concurrent` property.

Definition 8 (Failure)**.** *A declaration of* `failure E` *on feature* `f` *with property* `concurrent N` *means that* `f` *will immediately throw an exception of type* `E` *when the number of threads executing* `f` *is* `N` *and a new thread (one that is not already executing* `f`*) attempts to execute* `f`*.*

[4] Colloquially, we have been referring to this behavior as *bail-fast*.

By default, a feature has no `failure` property. It is erroneous for a feature to have multiple `failure` properties, as there can be only one exception thrown. A reimplemented feature has the same `failure` property as the original feature by default and can redeclare the property to narrow the failure exception type; this states that the particular implementation throws a more restricted exception if it fails. It is not permitted to add a failure property to a reimplementation of a feature where the original feature did not have one.

6.3.6 Atomic

The `atomic` property is used to informally state that a feature is *serializable*, i.e., that its behavior adheres to some definition of atomicity in concurrent systems. It does not enforce a particular definition of atomicity, and is primarily used to indicate that a suitable synchronization strategy, whether based on locks or on lock-free constructs, must be used in any implementation of the feature to ensure its serializability. A reimplemented feature inherits the `atomic` property.

6.3.7 Special

The `special` property is used to informally state additional concurrency information that cannot be captured by the other properties. It provides a natural-language description of the concurrent behavior of a class or feature. Multiple uses of `special` on a single construct are permitted, and it can be freely mixed with the other concurrency semantic properties. A reimplemented feature inherits any special properties of the original feature, and may declare additional special properties.

6.4 Refinement of Concurrency Properties

In general, when refining from an EBON specification to an implementation language, the concurrency properties described in the previous section are retained essentially unchanged[5] as semantic properties on the implementation constructs. This refinement can be applied in either direction, i.e., it is *reversible*. In particular, when refining to Java, concurrency properties may become specially-formatted comments; for example, `concurrency guarded 8` would refine to the Javadoc-like comment `// @concurrency guarded 8`. They may also become Java annotations, such as `@Concurrency("guarded 8")`, which are understood by modern Java compilers. The specific details of the notation for the semantic properties in Java are not critical for the purposes of this discussion.

In this section, we describe how these semantic properties are further refined to JML constructs for verification and runtime checking. We use JML2 syntax, where JML specifications are written as specially-formatted Java comments, throughout the following sections. While JML tools that support a more modern annotation-based JML syntax are under development, the JML2 syntax is much better supported at present.

This refinement mapping only looks straightforward in retrospect. The reader undoubtedly knows that there are often a number of subtle issues in the design of any domain specific language, and the concurrency semantic property is no exception. Balancing several opposing forces—primitive expressiveness, refinement coverage to the typical set of programming language and concurrency library constructs, modeler and developer use cases—is a difficult, delicate act.

[5] They are only changed when required by the particular target language, such as by replacing `Void` in EBON by `null` in Java.

We previously described the existing JML concurrency constructs, as well as additional constructs proposed by Rodríguez et al. [83] Of these, we use the following to transform our concurrency semantic properties to JML: the `when` clause and `commit` label, which specify that execution of a method suspends at a certain point until certain conditions are met; the `\lockset` expression, which returns the set of locks held by the current thread; and the `locks` expression, which specifies an upper bound on the set of locks that can be obtained by a thread executing a method.

We also need four additional constructs: an expression that indicates whether or not an object (or class, in the case of specifications on `static` methods) is broken as described in subsection 6.3.1; an expression that provides access to the implicit per-object per-method (or per-class per-method, for static methods) semaphores described in subsubsection 6.3.4.1; an expression that provides access to a count of the threads that have entered and not yet exited a given method on a given object (or class, for static methods); and an expression that provides access to the limit on the number of threads that may safely execute a method.

For the purposes of this discussion we will assume that these four new constructs are defined as JML primary expressions, described below. In an actual implementation, they might be added to JML in this way and supported with a runtime library, or they might be implemented as, e.g., model methods and fields on class `java.lang.Object`.

In the following definitions, a *descriptor* is a unique representation of a method's signature; for our purposes, this representation is a `String`.

Definition 9 (Broken Expression)**.** *`\broken` is a JML primary expression that returns a `boolean`. When used in an instance specification, it returns `true` if the object or its class is broken and `false` otherwise. When used in a static specification, it returns `true` if the class is broken and `false` otherwise.*

Recall that an object is broken when the `concurrent` property of one of its instance methods has been violated. Similarly, a class is broken if the `concurrent` property of one of its static methods, or a static method of a parent class, has been violated.

Definition 10 (Semaphore Expression)**.** *`\semaphore`, also usable with an optional descriptor as `\semaphore(descriptor)`, is a JML primary expression that returns an implicit per-object (or per-class) semaphore of class `java.util.concurrent.Semaphore`. When used without a descriptor in a method specification, it returns the per-object (or per-class) semaphore for that method (or static method), and when used without a descriptor outside a method specification it returns `null`. When used with a descriptor, it returns the per-object (or per-class) semaphore for the described method (or static method).*

Definition 11 (Thread Count Expression)**.** *`\thread_count` is a JML primary expression that returns an `int`. When used in a method specification, it returns the number of threads that are currently executing the method (for a given object if an instance method; in total if a static method). When used outside a method specification, it returns `0`.*

Definition 12 (Thread Limit Expression)**.** *`\thread_limit`, also usable with an optional descriptor as `\thread_limit(descriptor)`, is a JML primary expression that returns a non-negative `int`. When used without a descriptor in a method specification, it returns the maximum number of threads that may concurrently execute the method (for a given object if an instance method; in total if a static method), and when used without a descriptor outside of a method specification it returns `0`. When used with a descriptor, it returns the per-object (or per-class) thread limit for the described method (or static method).*

```
1  public class ThreadCounter {
2    //@ public initially active_count == 0;
3    //@ public invariant 0 <= active_count;
4    private /*@ spec_public @*/ int active_count;
5
6    //@ public invariant 0 < limit;
7    public final int limit;
8
9    //@ public initially !broken;
10   //@ public invariant limit < active_count ==> broken;
11   //@ public constraint \old(broken) ==> broken;
12   private /*@ spec_public @*/ boolean broken;
13
14   /*@ public invariant (\forall int i; counts.containsValue(i);
15     @                   0 < i); @*/
16   private /*@ spec_public @*/ final Map<Thread,Integer> counts;
17
18   /*@ ensures \old(counts.get(the_thread) == null) ==>
19     @           counts.get(the_thread) == 1 &&
20     @           active_count == \old(active_count + 1); */
21   /*@ ensures \old(counts.get(the_thread) != null) ==>
22     @           counts.get(the_thread) ==
23     @             \old(counts.get(the_thread) + 1); */
24   public synchronized void threadEnter(final Thread the_thread);
25
26   //@ requires counts.get(the_thread) != null;
27   //@ requires 0 < counts.get(the_thread);
28   /*@ ensures \old(1 < counts.get(the_thread)) ==>
29     @           counts.get(the_thread) ==
30     @             \old(counts.get(the_thread) - 1); */
31   /*@ ensures \old(counts.get(the_thread) == 1) ==>
32     @           counts.get(the_thread) == null &&
33     @           active_count == \old(active_count - 1); */
34   public synchronized void threadLeave(final Thread the_thread);
35 }
```

Listing 6.1: A sketch of the ThreadCounter class in Java.

At runtime, the `\broken` and `\thread_count` expressions are evaluated by counting the threads that execute methods and tracking whether the thread counts have exceeded any specified thresholds, and the `\semaphore` expression is evaluated by consulting a per-object (or per-class) map from methods to semaphores. To count all threads using methods of a class `C` at runtime we use one thread counter per instance method of `C` (including inherited methods) per instance of `C`, as well as an additional thread counter for each static method of `C`.[6] `\broken` holds for class `C` when any of the thread counters on static methods of `C` report that their bounds have been exceeded, and for an instance of `C` when any of that instance's method thread counters report that their bounds have been exceeded or when class `C` is broken. Listing 6.1 is a sketch (including JML specifications) of a thread counter class that could be used for this purpose.

Using this augmented set of JML concurrency constructs, we map each of our concurrency properties to JML and Java. In EBON, a concurrency property on a class applies to every feature of the class; when we refine a class with such a property to Java, the property applies to all methods of the Java class that are refined from features of the EBON class. EBON features with concurrency properties other than `special` always refine to Java methods rather than Java fields,

[6] In an actual implementation we could eliminate some overhead by not creating thread counters for methods where counting the threads is unnecessary, such as those declared `concurrency` ∞ that have no `failure` specifications and no JML specifications referring to `\thread_count`.

```
1  class C {
2    //@ invariant I;
3
4    //@ requires P; assignable A; ensures Q;
5    // @concurrency concurrent 4
6    public void m() {}
7  }
8
9  class C {
10   //@ invariant !\broken ==> I;
11
12   /*@ normal_behavior
13     @   requires P; assignable A; ensures !\broken ==> Q;
14     @ also normal_behavior
15     @   requires \thread_count == \thread_limit; assignable A;
16     @   ensures \broken; */
17   public void m() {}
18
19   /*@ invariant \typeof(this) == \type(C) ==>
20     @             \thread_limit("m()") == 4; */
21   //@ invariant 4 <= \thread_limit("m()");
22 }
```

Listing 6.2: Mapping of the `concurrent` property to JML.

because compliance with such properties cannot be reasonably enforced for field accesses. The mappings of our properties to JML do not discard any information and are therefore reversible.

6.4.1 Concurrent

The `concurrent` property places a restriction on the number of threads that can concurrently access a feature before its postcondition and the object invariant are no longer guaranteed. Using `\thread_count` and `\thread_limit` to determine when the thread limit has been reached, we map the `concurrent` property to JML as follows.

First, in any class `C` that has a finitely-bounded `concurrent` property on any method (including inherited methods), we prepend every invariant, constraint, and method postcondition with the precedent `!\broken ==>`. This indicates that the invariants and constraints are conditional on the object (or class) not being broken. We also add two invariants to `C` for each method `m` with a `concurrent` property to ensure that the thread limit for `m` is set and bounded appropriately and that child classes can increase it. The first invariant has the form
`N <= \thread_limit("m")`, and the second has the form
`\typeof(this) == \type(C) ==> \thread_limit("m") == N`.

In addition, if no `failure` property is specified, we add a specification case containing precondition `\thread_count == \thread_limit` and postcondition `\broken` to every method annotated with `concurrent N`. Therefore, if `N` threads are already executing the method, the next thread to enter must break the object (or class, depending on context). Any frame conditions in the original specification also apply to the added specification case, as exceeding a thread limit does not enable the method to modify state that it would otherwise have been unable to modify. Listing 6.2 shows an example of this mapping.

```
1  class C {
2    //@ requires P; assignable A; ensures Q;
3    // @concurrency guarded A, B, 3
4    public void m() {
5      // method body
6    }
7  }
8
9  class C {
10   //@ requires P; assignable A; ensures Q;
11   /*@ when \lockset.has(A) & \lockset.has(B) &
12     @      \lockset.has(\semaphore); */
13   /*@ ensures (\lockset.has(A) <==> \old(\lockset.has(A))) &
14     @         (\lockset.has(B) <==> \old(\lockset.has(B))) &
15     @         (\lockset.has(\semaphore) <==>
16     @           \old(\lockset.has(\semaphore))); */
17   public void m() {
18     commit:
19     // method body
20   }
21   //@ initially \semaphore("m()").permits() == 3;
22 }
```

Listing 6.3: Mapping of the `guarded` property to JML.

6.4.2 Locks

The `locks` property is an exact analogue of the JML `locks` clause proposed by Rodríguez et al. [83], which lists the maximal set of locks that can be acquired by a thread while executing a method. Thus, to map a `locks` property on a method to JML, we simply add a JML `locks` clause with the same list of locks to every specification case of the method.

6.4.3 Guarded

The `guarded` property strictly specifies the locking and blocking behavior of a feature. Each listed lock, including the implicit semaphore for the feature if applicable, must be held by a thread before it executes the feature; moreover, upon exiting the feature, a thread must not hold any locks it acquired upon entry to the feature to satisfy the `guarded` property.

We map the `guarded` property to JML by adding the following specifications to each `guarded` method: a `locks` clause listing the locks from the `guarded` property; a `when` clause indicating that all listed locks must be held in order for the method to proceed; a `commit` label before the first statement of the method; and a postcondition for each listed lock that states the lock is held on exit if and only if it was held on entry. For each method that uses an implicit semaphore, we also add an `initially` clause to the class to specify the initial value of that implicit semaphore as provided in the `guarded` property. Listing 6.3 shows an example of this mapping.

6.4.3.1 `synchronized` in Java

The property `guarded Current` is a special case when refined to Java, because the lock on the current object (`this`) in Java is only accessible through the `synchronized` keyword. A feature declared `guarded Current` therefore must always refine to a Java `synchronized` method; this is the case even when refining to Java with semantic properties rather than full JML, since the

```
1  class C {
2    //@ requires P; assignable A; ensures Q;
3    // @concurrency concurrent 8
4    // @concurrency failure E
5    public void m() {}
6  }
7
8  class C {
9    /*@ normal_behavior
10     @   requires \thread_count < \thread_limit;
11     @   requires P; assignable A; ensures \broken ==> Q;
12     @ also exceptional_behavior
13     @   requires \thread_count == \thread_limit;
14     @   assignable \nothing; signals E true; signals_only E; */
15   public void m() throws E {}
16
17   /*@ invariant \typeof(this) == \type(C) ==>
18         \thread_limit("m()") == 8; */
19   //@ invariant 8 <= \thread_limit("m()");
20 }
```

Listing 6.4: Mapping of the `failure` property to JML.

`synchronized` keyword is part of the method's API and cannot reasonably be added when later mapping the semantic properties to JML.

The locking discipline Java imposes on `synchronized` methods exactly matches the locking discipline we impose on `guarded` features, so no additional JML specification is necessary. Since `guarded` is an inherited property and the Java `synchronized` modifier is not inherited, any override of such a method must also be explicitly declared `synchronized` to conform to the `guarded` specification.

6.4.4 Failure

The `failure` property specifies an exception that will be thrown in any thread attempting to exceed the `concurrent` bound of a feature. Using `\thread_count` and `\thread_limit` to determine when the thread limit has been reached, we map the `failure` property to JML as follows.

For every method with a `failure` property, we add a precondition to every specification case of the method stating that the thread count must be within the specified limit. Then, we add an `exceptional_behavior` specification case to the method with a precondition that the thread count is at the limit, a frame condition that says no state will be changed, and `signals` and `signals_only` clauses that specify the exception to be thrown. Thus, all the original specification cases of the method remain in force for threads that are within the limit and every thread that attempts to exceed the limit triggers the exception. Finally, we add a `throws` clause to the method to indicate that it can throw the failure exception. The `throws` clause is added even when refining to Java with semantic properties rather than full JML, since it is part of the method's signature and cannot reasonably be added when later mapping the semantic properties to JML. Listing 6.4 shows an example of this mapping.

6.4.5 Atomic

The `atomic` property is an exact analogue of the JML `atomic` method modifier proposed by Rodríguez et al. [83], which indicates that the execution of the method is serializable. Thus, we

add a JML `atomic` modifier to any method refined from an `atomic` feature. Rodriguez et al. do not discuss how inheritance and atomicity interact; we follow Flanagan et al. [34] in making atomicity covariantly inherited, as discussed in subsection 6.3.6.

6.4.6 Special

The `special` property contains a textual description of a concurrency strategy, which cannot be directly mapped to JML or Java except as a semantic property realized as a comment or annotation. Therefore, `special` properties are retained as-is in a refined Java program and are used to guide the implementation.

6.5 Example

We illustrate the use of the concurrency semantic property by applying it to solutions of two classic synchronization problems: Dijkstra's dining philosophers problem and the sleeping barber problem. We show the formal BON specifications and the refined JML-annotated Java class skeletons for each problem, omitting the high-level English specifications and the Java implementations to conserve space.

6.5.1 Dining Philosophers

In the dining philosophers problem, *N* philosophers (Dijkstra specified 5, but the problem generalizes easily) are gathered around a circular table. Each philosopher has identical behavior (that is, the code for their synchronization routines is identical). At all times, each philosopher is either *thinking* or *eating*. A *thinking* philosopher may choose to remain *thinking* forever (and thereby starve), or may at some point become *eating*. An *eating* philosopher must become *thinking* in finite time.

A fork sits on the table between each pair of adjacent philsophers; thus each philosopher has a left fork and a right fork, and the number of forks is equal to the number of philosophers. In order to become *eating*, a philosopher must hold both his left and right forks; moreover, each philosopher is permitted to hold *only* his left and right forks and no other forks. The synchronization problem is to structure the behavior of the system to maintain this fork discipline (the safety condition) while ensuring the (relatively weak) progress condition that, whenever any philosopher is waiting to obtain a fork (i.e., the fork is held by another philosopher), *some* philosopher is *eating*. Clearly, the system can fail to work in several ways; for instance, a deadlock can occur when each philosopher is holding one fork (e.g., they have all picked up their left forks) and waiting forever for a second fork that will never become available.

Many solutions to the dining philosophers problem exist. One simple solution is to impose a total order on the forks; for example, assign integers to forks starting at 0 for the left fork of the first philosopher and increasing around the table until the left fork of the last philosopher (who has fork 0, the bottom of the total order, as his right fork). Then, require each philosopher to pick up his lower-numbered fork first and his higher-numbered fork second. This solution works because it breaks the symmetry of the problem: most of the philosophers will pick up their left forks first, because their left forks will be lower in the order, but the last philosopher will pick up his right fork first because his right fork is the bottom of the order. The situation where every philosopher is holding one fork cannot occur, because that would require each philosopher to hold the fork he picked up first and two of them cannot hold the bottom fork simultaneously.

We model this solution using three EBON classes, shown in Listing 6.5: `FORK`, `PHILOSOPHER`, and **`MAIN`**. `FORK` is a semaphore with initial value 1, which is indicated by the postcondition on its `make` feature; by EBON convention, the `make` feature is a constructor. `PHILOSOPHER` models a philosopher that has left and right forks, a `run` feature, and lifecycle features `think` (the non-critical section) and `eat` (the critical section). **`MAIN`** constructs the forks and philosophers, assigns forks to philosophers such that the philosophers form a circle (the postcondition of `make`), and starts the philosopher threads.

The concurrency of this system is specified primarily using `guarded` features. `think` and `eat` both have concurrency specification `guarded Current`, which ensures mutual exclusion between them. `eat` also has concurrency specification `guarded left_fork, right_fork`, which means that both forks and the lock on `Current` must be obtained before or upon its invocation. The lock ordering invariant on **`MAIN`** ensures that the order in which the forks are obtained is, in fact, a total order as described above. The lock ordering invariant on `PHILOSOPHER` ensures that the lock on `Current` is always obtained before the forks; this prevents a deadlock that could occur if `eat` and `think` are invoked simultaneously, the thread invoking `think` obtains the lock on `Current` and never terminates, and the thread invoking `eat` obtains one or both forks and never releases them because it never obtains the lock on `Current`. `run`, which repeatedly executes `think` and `eat`, has concurrency specification `guarded Current` to indicate that the philosopher can only be run by one thread a time and to ensure that, when the philosopher is running, only the thread that invoked `run` can invoke `eat` and `think`. It also has concurrency specification `locks left_fork , right_fork` to indicate that it can obtain the forks during its execution (in this case, by calling `eat`).

Listing 6.6 shows a refinement of the EBON classes to JML-annotated Java with concurrency semantic properties. In this refinement, we have included almost enough Java code for the system to run; the only parts missing are "obvious" methods (i.e., some constructors), the code that acquires and releases the semaphores in the proper order, and the implementations of the critical and non-critical sections. The `Semaphore` class extended by `Fork` is `java.util.concurrent.Semaphore`, which implements a counting semaphore; the postcondition of the `Fork` constructor provides its initial value of 1. The `guarded Current` properties on `eat`, `think`, and `run` refine to Java `synchronized` keywords. The `guarded left_fork, right_fork` property on `eat` and the `locks` property on `run` are carried as-is into the refined Java code. When later mapped to JML the specification on `eat` refines as described in subsection 6.4.3 to several JML clauses, shown in Listing 6.7. The `locks` specification on `run` refines directly to a JML `locks` clause listing the same locks.

One simple solution to the problem is C.S. Scholten's *butler solution*. In this solution, each fork has its own binary semaphore and there is an additional counting semaphore, the butler semaphore, with a bound of $N-1$. The basic idea is that each philosopher, when it decides to eat, acquires first the butler semaphore, then the semaphore for its left fork, and finally the semaphore for its right fork. Deadlock is prevented because the butler semaphore ensures that only $N-1$ of the philosophers can attempt to acquire forks simultaneously; since there are N forks, at least one of the philosophers must succeed and be able to eat.

We implement this solution using four EBON classes, shown in Listing 6.8: `BUTLER`, `FORK`, `PHILOSOPHER` and **`MAIN`**. `BUTLER` and `FORK` are counting semaphores. `make`, by EBON convention, is a constructor, and the constraint on the `initial_value` feature in the postcondition of `BUTLER.make` indicates that it is setting the upper bound of the semaphore. `BUTLER` has the concurrency semantics `guarded (num_phils $-$ 1)`, which means that the butler can be concurrently acquired by at most one fewer than the number of philosophers it is initialized with; `FORK` has the concurrency semantics `guarded 1`, which means that a fork can only be acquired by one philosopher at a time. The philoso-

pher's `eat` feature has the concurrency semantics `guarded butler, left_fork, right_fork` (shorthand for three separate `guarded` statements, for space reasons), meaning that it acquires those three locks before execution.

A refinement of this solution to JML-annotated Java class skeletons appears in Listing 6.9; we have omitted the refinement of the `MAIN` class for space reasons. The `Semaphore` class extended by `Butler` and `Fork` is `java.util.concurrent.Semaphore`, which implements a counting semaphore; the bound of the `Butler` semaphore is `my_num_phils − 1` and the bound of the `Fork` semaphore is 1, as in the original solution. Note that this is not the only possible refinement. The additional features of `Butler` and `Fork` are only used in the class invariant, and there they are used to redundantly specify lock behavior (i.e., if a philosopher's thread holds the forks, then the forks know that philosopher holds them). If we were willing to destroy the dining philosophers metaphor, we could eliminate the `Butler` and `Fork` classes entirely and replace them with `Semaphore`.

6.5.2 Sleeping Barber

The sleeping barber problem involves a barber shop with one barber, one barber chair, and a waiting room with N chairs. If a customer arrives and the barber chair is occupied, the customer sits in a waiting room chair if one is available and leaves otherwise (without getting a haircut); the customer continues to try to get a haircut until he is allowed into the shop. If the barber finishes giving a haircut (that is, the customer in the barber chair leaves) when there are customers in the waiting room, the barber puts one of them in the barber chair and cuts his hair; otherwise, the barber goes to sleep (hence the problem's name). If a customer arrives and the barber chair is empty, the customer wakes the barber and sits in the barber chair. The synchronization problem is to maintain the progress property that the barber is continuously cutting hair as long as there are customers and sleeping (i.e., inactive, as opposed to busy waiting) in the absence of customers, while preserving the safety property that there are at most N customers in the waiting room and 1 customer getting a haircut.

One common solution to the problem models the barber and the customers as individual processes sharing three semaphores and an integer k, initially with value N, that represents the number of empty seats. One semaphore C, unbounded and starting at 0, counts the customers waiting in the waiting room; one binary semaphore B, also starting at 0, indicates when the barber is ready to cut hair; and one binary semaphore X, starting at 1, provides mutually exclusive access to the number of empty seats so that it can be safely modified. The barber continually repeats the following sequence of operations: acquire C (get a customer from the waiting room); acquire X (get exclusive access to k); increment k (free the customer's seat); release B (tell the customer it's time for a haircut); release X; cut hair. Each customer repeats the following sequence of operations: acquire X; if there are no free seats, release X and try again later from the beginning, otherwise continue; decrement k (sit down); release C (notify the barber); release X; acquire B (wait for the barber); get hair cut.

Our solution is similar to this one in principle. However, we eliminate all the explicit semaphore manipulation and the counter by using concurrency properties; Listing 6.10 shows our EBON specification of the system. Our barber is a passive, rather than active, process. It has the single feature `cut_hair` and is considered to be sleeping whenever this feature is not in use, which eliminates the need for the C semaphore. `cut_hair` has concurrency `guarded Current`, meaning that the barber may cut only one customer's hair at a time and other customers attempting to get haircuts will be suspended; this replaces the B semaphore. The `concurrent` and `failure` properties on **`BARBER_SHOP`**`.get_hair_cut` handles the waiting room chairs, replacing the X semaphore

and the counter. A maximum of `num_seats + 1` customers may use this feature concurrently. The customers who are not actually getting haircuts (because the barber can only handle one customer at a time) are considered to be waiting in chairs, and any additional customers beyond the limit immediately receive a **`NO_SEATS`** exception when they attempt to get haircuts. The entire `CUSTOMER` class has concurrency `guarded Current` to indicate that all its features lock `Current`; this ensures that only one thread can be accessing features of **`CUSTOMER`** at a time, and therefore that the value of `needs_hair_cut` is always changed atomically. Various features have `delta` contracts as frame conditions to indicate the features they can change the values of when invoked; for example, **`BARBER`**`.cut_hair` can change the value of `needs_hair_cut` on its **`CUSTOMER`** parameter.

Listing 6.11 shows a refinement of the EBON classes to Java with concurrency semantic properties, and Listing 6.12 and Listing 6.13 shows a further refinement where all the concurrency semantic properties are refined to JML and almost enough Java code is included for the system to run. As a reminder, while the top-level EBON specification from Listing 6.10 would be manually written by the system designer, these refinements would be automatically generated from the EBON classes. For space reasons, we have omitted the contents of some "obvious" methods (i.e., constructors and accessors), as well as the `Main` and `NoSeats` classes. In `Barber`, the `guarded Current` property on `cut_hair` refines to `synchronized`; in `Customer`, the `guarded Current` property on the class refines to the use of `synchronized` on all instance methods. In `BarberShop`, the `concurrent 4` and `failure` **`NO_SEATS`** properties on `get_hair_cut` refine to JML as described in Sections 6.4.1 and 6.4.4. Thus, when there are still waiting chairs available the caller will enter the method (and immediately execute, or suspend on, `cutHair()`), and when the maximum number of customers is already waiting the caller will receive a `NoSeats` exception.

6.6 Conclusions and Future Work

The use of a concurrency design discipline such as we have described should not significantly impact the performance of the resulting system. Runtime checking against thread limits necessarily incurs time and space overhead, and thus will have some impact, which we expect to be minimal. Additionally, many runtime checks can be eliminated with appropriate use of verification tools [93].

Tool support is necessary to check the correctness of the use of the `concurrency` property at the design level, the correctness of the refinement to JML, and the correctness of an implementation with respect to its concurrency specification.

Previous to this work, we have reasoned about concurrent Java systems using proof-carrying code within the EU FP6 "Mobius" project for the past four years.[7] Within this project, primitive concurrency annotations were written at the source level in "model-less" JML and, as such, we wrestled with many challenges involving low-level concurrency primitives like threads, locks, static initialization, construtor semantics, and the Java memory model. This challenging and sometimes frustrating program verification experience was part of the motivation for the development of these model-level annotations.

A tool-independent plugin library, code-named "Propi," which can be integrated into existing and future tools such as the Common JML tool suite, ESC/Java2, and OpenJML, is under development.[8] Complementing this implementation, a mechanization in higher-order logic, including

[7] See http://mobius.inria.fr/.

[8] See http://kindsoftware.com/products/opensource/Propi/.

a proof of safety preservation during refinement, is underway.[9]

One aspect of concurrent programming that we have not yet described with the concurrency semantic property, and that has not been widely addressed in recent literature, is *fairness*, the absence (or limitation) of resource starvation in a system. For example, the solution we presented to the sleeping barber problem in the previous section exhibits two types of unfairness: first, a customer who finds the barber shop full may return and find it full again even if another customer has entered and taken a waiting chair in the interim; second, a customer who enters the shop and takes a waiting chair may never receive a haircut because the Java monitor on the `synchronized` method `cutHair()` makes no ordering guarantees with respect to waiting threads. Thus, a particularly unlucky customer could sit in a waiting chair forever watching new customers walk in, sit down, and get haircuts.

The first type of unfairness is inherent to the problem statement and cannot be remedied without requiring customers to queue on the sidewalk outside the shop. The second, however, can certainly be fixed: if the `cutHair()` guard was a fair lock (such as a `java.util.concurrent.ReentrantLock` with fairness turned on), the customers in the shop would effectively be queuing for the barber, at the cost of some synchronization performance. One extension to this work that we intend to investigate is the specification of fairness in high-level models, through the addition of a fair variant to the existing `guarded` property and new or modified notation to describe the fairness of explicitly-declared locks. In general, Cargill's *Specific Notification* pattern [17], which is similar to Martin's *airlock* construct [74], can be used to guarantee fairness of `guarded` features.

Reasoning about correctness in the presence of concurrency and the absence of a locking discipline, as is the case when no `locks` clauses are used, is still an open problem. Huisman and Hurlin state stability conditions under which such reasoning is sound [46], and Nienaltowski et al. present a sound reasoning technique in the context of the Eiffel programming language [79].

We believe it is possible to build a "Daikon [25] of concurrency" to detect common synchronization patterns and generate model-level concurrency annotations. The main challenge in such work is to infer models from ad hoc uses of concurrency, much like Flanagan and his colleagues have been doing for several years [2].

Finally, as we have witnessed in the several static analysis tools and techniques we have developed, statically reasoning about object-oriented systems in the presence of polymorphism and dynamic binding, while challenging, is tractable. We have found that good developers do not abuse these mechanisms unnecessarily; thus, while we can construct nefarious examples that stump even rich semantic-based static analysis techniques like those used in our earlier tools, such examples are rare in practice. Recent research indicates that this pattern seems to continue in concurrent software [34].

[9] See http://kindsoftware.com/documents/mech_theory/concurrency_semantic_property.html.

```
class FORK
  inherit SEMAPHORE
  feature
    number: INTEGER
    make
      -> the_number: INTEGER
      ensure number = the_number; initial_value = 1
    end
end

class PHILOSOPHER
    left_fork: FORK
    right_fork: FORK
    eat
      concurrency guarded Current, left_fork, right_fork
      -- behavior: user-defined critical section
    end
    think
      concurrency guarded Current
      -- behavior: user-defined non-critical section
    end
    run
      concurrency guarded Current;
                  locks left_fork, right_fork
      -- behavior: repeatedly think; eat
    make
      -> the_left_fork, the_right_fork: FORK
      require the_left_fork /= the_right_fork;
      ensure left_fork = the_left_fork;
             right_fork = the_right_fork
    end
  invariant
    Current <# left_fork; Current <# right_fork
end

class MAIN
  feature
    forks: SEQUENCE[FORK]
    phils: SEQUENCE[PHILOSOPHER]
    make
      -> the_count: INTEGER
      ensure forks.length = the_count;
             phils.length = the_count;
             for_all i, j: INTEGER
             such_that 0 <= i and i < the_count and
                       0 <= j and j < the_count
             it_holds
               phils[i].left_fork = forks[i] and
               phils[i].right_fork =
                 forks[(i + 1) \\ the_count] and
               forks[i].number = i and
               forks[i] = forks[j] <-> i = j
      -- behavior: create the_count forks in the sequence;
      -- create the_count philosophers, giving each two forks
      -- such that the postcondition is satisfied; "run" each
      -- philosopher in its own thread
    end
  invariant
    for_all i, j: INTEGER
    such_that 0 <= i and i < j and j < forks.length
    it_holds forks[i] <# forks[j]
end
```

Listing 6.5: EBON classes for the dining philosophers problem.

```
1  public class Fork extends Semaphore {
2    public final int number;
3
4    //@ ensures number = the_number;
5    //@ ensures availablePermits() == 1;
6    public Fork(final int the_number) {
7      super(1); number = the_number;
8    }
9  }
10
11 public class Philosopher {
12   private /*@ spec_public @*/ final Fork left_fork;
13   private /*@ spec_public @*/ final Fork right_fork;
14
15   //@ requires the_left_fork != the_right_fork;
16   //@ ensures left_fork == the_left_fork
17   //@ ensures right_fork == the_right_fork;
18   public Philosopher(final Fork the_left_fork,
19                      final Fork the_right_fork) {}
20
21   // @guarded left_fork, right_fork
22   public synchronized void eat() {
23     // critical section
24   }
25
26   public synchronized void think() {
27     // non-critical section
28   }
29
30   // @locks left_fork, right_fork;
31   public synchronized void run() {
32     while (true) { think(); eat(); }
33   }
34
35   //@ public invariant this <# left_fork & this <# right_fork;
36 }
37
38 public class Main {
39   private /*@ spec_public @*/ final Fork[] forks;
40   private /*@ spec_public @*/ final Philosopher[] phils;
41
42   //@ ensures forks.length = the_count;
43   //@ ensures phils.length = the_count;
44   /*@ ensures (\forall int i, j;
45                0 <= i & i < the_count & 0 < j & j < the_count;
46                phils[i].left_fork == my_forks[i] &
47                phils[i].right_fork == forks[(i + 1) % the_count] &
48                forks[i].number == i &
49                forks[i] == forks[j] <==> i == j); @*/
50   public Main(final int the_count) {
51     // create everything and start threads
52   }
53
54   /*@ public invariant
55       (\forall int i, j: 0 <= i & i < j & j < forks.length;
56        forks[i] <# forks[j]); @*/
57 }
```

Listing 6.6: A refinement of the dining philosophers classes to Java with concurrency properties.

```
1  //@ locks left_fork, right_fork;
2  //@ when \lockset.has(left_fork) & \lockset.has(right_fork);
3  /*@ ensures  (\lockset.has(left_fork) <==>
4    @           \old(\lockset.has(left_fork)) &
5    @          (\lockset.has(right_fork) <==>
6    @           \old(\lockset.has(right_fork))); */
7  public synchronized void eat() {
8    // obtain the forks in order
9    /*@ commit: @*/
10   // critical section
11   // release the forks
12 }
```

Listing 6.7: A refinement of the `eat` method specification to JML.

```
class BUTLER inherit SEMAPHORE
  feature
    make
      -> the_num_phils: INTEGER
      require 0 < the_num_phils
      ensure initial_value = the_num_phils - 1
    end
end

class FORK inherit SEMAPHORE
  feature
    make
      ensure initial_value = 1
    end
end

class PHILOSOPHER
  feature
    butler: BUTLER
    left_fork: FORK
    right_fork: FORK
    eat
      concurrency guarded butler, left_fork, right_fork, Current
      -- behavior: user-defined critical section
    end
    think
      concurrency guarded Current
      -- behavior: user-defined non-critical section
    end
    run
      concurrency locks butler, left_fork, right_fork; guarded Current
      -- behavior: repeatedly think; eat
    end
    make
      -> the_butler: BUTLER
      -> the_left_fork, the_right_fork: FORK
      require the_left_fork != the_right_fork;
      ensure butler = the_butler;
             left_fork = the_left_fork;
             right_fork = the_right_fork
    end
end

class MAIN
  feature
    forks: SEQUENCE[FORK]
    phils: SEQUENCE[PHILOSOPHER]
    make
      -> the_count: INTEGER
      ensure forks.length = the_count;
             phils.length = the_count;
             for_all i, j: INTEGER
             such_that 0 <= i and i < the_count and 0 <= j and j < the_count
             it_holds
               phils[i].left_fork = forks[i] and
               phils[i].right_fork = forks[(i + 1) \\ the_count] and
               forks[i] = forks[j] <-> i = j
      -- behavior: create the_count forks in the sequence;
      -- create the_count philosophers, giving each two forks
      -- such that the postcondition is satisfied; "run" each
      -- philosopher in its own thread
    end
  invariant
    for_all i: INTEGER such_that 0 <= i it_holds butler <# forks[i]
end
```

Listing 6.8: EBON classes for the dining philosophers problem.

```
1  public class Butler extends Semaphore {
2    //@ require 0 < the_num_phils;
3    //@ ensures availablePermits() == the_num_phils - 1;
4    public Butler(final int the_num_phils) {
5      super(the_num_phils - 1);
6    }
7  }
8
9  public class Fork extends Semaphore {
10   //@ ensures availablePermits() == 1;
11   public Fork() { super(1); }
12 }
13
14 public class Philosopher {
15   private /*@ spec_public @*/ final Butler butler;
16   private /*@ spec_public @*/ final Fork left_fork;
17   private /*@ spec_public @*/ final Fork right_fork;
18
19   //@ requires the_left_fork != the_right_fork;
20   //@ ensures butler == the_butler;
21   //@ ensures left_fork == the_left_fork
22   //@ ensures right_fork == the_right_fork;
23   public Philosopher(final Butler the_butler,
24                      final Fork the_left_fork,
25                      final Fork the_right_fork) {}
26
27   //@ locks butler, left_fork, right_fork;
28   // @concurrency guarded butler, left_fork, right_fork;
29   protected synchronized void eat() {
30     // user-defined critical section
31   }
32
33   public synchronized void think() {
34     // user-defined non-critical section
35   }
36
37   //@ locks butler, left_fork, right_fork;
38   public synchronized void run() {
39     while (true) { think(); eat(); }
40   }
41 }
42
43 public class Main {
44   private /*@ spec_public @*/ final Fork[] forks;
45   private /*@ spec_public @*/ final Philosopher[] phils;
46   private /*@ spec_public @*/ final Butler butler;
47
48   //@ ensures forks.length = the_count;
49   //@ ensures phils.length = the_count;
50   /*@ ensures (\forall int i, j;
51                0 <= i & i < the_count & 0 < j & j < the_count;
52                phils[i].left_fork == forks[i] &
53                phils[i].right_fork == forks[(i + 1) % the_count] &
54                phils[i].butler == butler &
55                forks[i] == forks[j] <==> i == j); @*/
56   public Main(final int the_count) {
57     // create everything and start threads
58   }
59
60   /*@ public invariant
61       (\forall int i: 0 <= i & i < forks.length;
62        butler <# forks[i]); @*/
63 }
```

Listing 6.9: A refinement of the dining philosophers classes to JML-annotated Java skeletons.

```
class BARBER
  feature
    cut_hair -> c: CUSTOMER
      concurrency guarded Current
      ensure not c.needs_hair_cut; delta c.needs_hair_cut
end

class BARBER_SHOP
  feature
    barber: BARBER
    num_seats: INTEGER
      ensure 0 < Result
    end
    get_hair_cut -> c: CUSTOMER
      concurrency concurrent (num_seats + 1)
      concurrency failure NO_SEATS
      concurrency locks barber
      ensure not c.needs_hair_cut; delta c.needs_hair_cut
      -- behavior: barber.cut_hair(c)
    end
    make
      -> the_barber: BARBER
      -> the_num_seats: INTEGER
      ensure barber = the_barber; num_seats = the_num_seats
    end
end

class CUSTOMER
  concurrency guarded Current
  feature
    shop: BARBER_SHOP
      ensure Result /= Void
    end
    needs_hair_cut: BOOLEAN
    set_hair_cut
      ensure not needs_hair_cut; delta needs_hair_cut
    end
    regular_activities
      require not needs_hair_cut
      ensure needs_hair_cut; delta needs_hair_cut
      -- behavior: whatever the customer does between haircuts
    end
    run
      concurrency locks shop.barber
      ensure delta needs_hair_cut
      -- behavior: repeatedly execute the sequence
      -- "regular_activities; retry shop.get_hair_cut until not needs_hair_cut"
    end
    make
      -> the_shop: BARBER_SHOP
     ensure shop = the_shop; not needs_hair_cut
    end
end

class MAIN
  feature
    make -> the_seats: INTEGER -> the_customers: INTEGER
      require 0 < the_seats; 0 < the_customers
      -- behavior: create a barber, create a barber shop with the_seats seats,
      -- create the_customers customers, and "run" each customer in a separate thread
    end
end

class NO_SEATS inherit EXCEPTION end
```

Listing 6.10: EBON classes for the sleeping barber problem.

```
1  public class Barber {
2    //@ assignable the_c.needsHairCut;
3    //@ ensures !the_c.needsHairCut();
4    public synchronized void cutHair(final Customer the_c) {}
5  }
6
7  public class BarberShop {
8    private /*@ spec_public @*/ final Barber barber;
9    private /*@ spec_public @*/ final int num_seats;
10
11   //@ requires 0 < num_seats;
12   //@ ensures barber == the_barber
13   //@ ensures num_seats == the_num_seats;
14   public BarberShop(final Barber the_barber,
15                     final int the_num_seats) {}
16
17   //@ assignable the_c.needsHairCut;
18   //@ ensures !the_c.needsHairCut();
19   // @concurrency concurrent (num_seats + 1)
20   // @concurrency failure NoSeats
21   // @concurrency locks barber
22   public void getHairCut(final Customer the_c)
23     throws NoSeats {}
24 }
25
26 // @concurrency guarded this
27 public class Customer {
28   private final BarberShop shop;
29   private boolean my_needs_hair_cut; //@ in needsHairCut;
30
31   //@ ensures shop() == the_shop;
32   //@ ensures !needsHairCut();
33   public Customer(final BarberShop the_shop) {}
34
35   public synchronized BarberShop shop() {}
36
37   //@ public model non_null JMLDataGroup needsHairCut;
38   //@ in objectState;
39   public synchronized boolean needsHairCut() {}
40
41   //@ assignable needsHairCut;
42   //@ ensures !needsHairCut();
43   public synchronized void setHairCut() {}
44
45   //@ requires !needsHairCut();
46   //@ assignable needsHairCut;
47   //@ ensures needsHairCut();
48   public synchronized void regularActivities() {}
49
50   //@ assignable needsHairCut;
51   // @concurrency locks shop().barber
52   public synchronized void run() {}
53 }
```

Listing 6.11: A refinement of the sleeping barber classes to Java with concurrency properties.

```
1  public class Barber {
2    //@ assignable the_c.needsHairCut;
3    //@ ensures !the_c.needsHairCut();
4    public synchronized void cutHair(final Customer the_c) {
5      // take some time to do it right
6      the_c.setHairCut();
7    }
8  }
9
10 public class Customer {
11   private final BarberShop my_shop;
12   private boolean my_needs_hair_cut; //@ in needsHairCut;
13
14   //@ ensures shop() == the_shop;
15   //@ ensures !needsHairCut();
16   public Customer(final BarberShop the_shop) {}
17
18   public synchronized BarberShop shop() {}
19
20   //@ public model non_null JMLDataGroup needsHairCut;
21   //@ in objectState;
22   public synchronized boolean needsHairCut() {}
23
24   //@ assignable needsHairCut;
25   //@ ensures !needsHairCut();
26   public synchronized void setHairCut() {}
27
28   //@ requires !needsHairCut(); assignable needsHairCut;
29   //@ ensures needsHairCut();
30   public synchronized void regularActivities() {
31     // whatever the customer does between haircuts
32     my_needs_hair_cut = true;
33   }
34
35   //@ assignable needsHairCut; locks shop().barber;
36   public synchronized void run() {
37     while (true) {
38       regularActivities();
39       while (my_needs_hair_cut) {
40         try { my_shop.getHairCut(); }
41         catch (final NoSeats ns) { /* insufficient seats */ }
42       }
43     }
44   }
45 }
```

Listing 6.12: A refinement of the sleeping barber classes to Java with full JML refinement of concurrency properties (classes `Barber` and `Customer`).

```
1  public class BarberShop {
2    private /*@ spec_public @*/ final Barber barber;
3    private /*@ spec_public @*/ final int num_seats;
4
5    //@ requires 0 < num_seats;
6    //@ ensures barber == the_barber;
7    //@ ensures num_seats == the_num_seats;
8    public BarberShop(final Barber the_barber,
9                      final int the_num_seats) {}
10
11   /*@ public normal_behavior
12     @   requires \thread_count < \thread_limit;
13     @   assignable the_c.needsHairCut;
14     @   locks barber;
15     @   ensures \broken ==> !the_c.needsHairCut();
16     @ also public exceptional_behavior
17     @   requires \thread_count == \thread_limit;
18     @   assignable \nothing;
19     @   signals NoSeats true; signals_only NoSeats; */
20   public void getHairCut(final Customer the_c) throws NoSeats {
21     barber.cutHair(the_c);
22   }
23
24   /*@ invariant \typeof(this) == \type(BarberShop) ==>
25     @             \thread_limit("getHairCut(Customer)") ==
26     @               num_seats + 1; */
27   /*@ invariant num_seats + 1 <=
28     @             \thread_limit("getHairCut(Customer)"); */
29 }
```

Listing 6.13: A refinement of the sleeping barber classes to Java with full JML refinement of concurrency properties (class `BarberShop`).

Chapter 7

Concurrency Models

Concurrency models come in three form in this work. Firstly, there are the normal, behavioral specifications of the Java concurrency API, discussed briefly here but covered in more detail in Deliverable D6.6. Second, there are mechanically specified 'native' model types for Java and JML. This type structure provides the foundation for reasoning about models and their inter-relationships.

7.1 JML Annotations for the Concurrency API

In their work in 2005, Rodriguez et al. annotated much of the concurrency API with their ECOOP paper annotations. Unfortunately, these specifications are no longer available.

Bloom and Huisman led the new specification of classes in the Java concurrency API, as reported in Chapter 4 of Deliverable D6.4. As their specifications focus on a combination of JML and permission-based separation logic, they are outside the scope of refinement defined in this deliverable.

The techniques that they evidence provide a foundation for later work integrating our model-based refinement approach with general-purpose higher-order separation logic frameworks, like the Charge! framework from Bengtson, Jensen, and Birkedal [8].

7.2 JML Model Types

We have developed a general-purpose set of Java and JML model types in higher-order logic, mechanized in the PVS interactive theorem prover. These model types have been used for several pieces of research, independent of our initial experiments in specifying and reasoning about Java concurrency in a refinement-centric, model-based setting.

We only mechanize the base minimum of constructs necessary to target refinement, as discussed in the previous chapter and in other publications [60]. They are of a similar flavor of the mechanizations witnessed in previous and current work on formal verification of Java programs from Jacobs et al. [50] and Birkedal et al. [8], respectively.

Draft formalizations are found in Appendix B. Included in these theories are all core Java notions down to the field and method declaration level (i.e., there is no mention of bodies or statements, as they are unnecessary) and many JML notions in JML level 0, JML level 1, and JML level 'C'. We refrain from over-cross-referencing to the relevant sections in the Appendices.

In slightly more detail, the theory `java` describes modules (classes and interfaces), types (both reference and primitive), exceptions, variables, visibility modifiers, field modifiers, fields,

and method declarations. The theories `java_inheritance_semantics` and `java_semantics` describe exactly the semantics inherent in the types introduced in the theory `java`. For example, inheritance is described, the meaning of `implements` versus `extends`, the inheritance structure of `Throwable`, etc.

Next, the theory `jml` introduces the core notions of assertions, frames, contracts, purity, ghost fields, model fields and methods, and invariants and history constraints.

These types and their axiomatic specification provide sufficient foundation for specifying a sketch HOL theory of Java concurrency.

7.3 A HOL Theory of Java Concurrency

All concurrency annotations discussed in the last chapter have also been formalized in PVS. The draft specifications are found in Appendix B. These theories are written in the context of the aforementioned theories for Java and JML.

The datatype `Concurrency` describes the `concurrency` semantic property. The parameterized theory `semantic_property` permits one to specify concurrency annotations for Java and JML types, and this property's defaults are captured in the theory `defaults`. The theory `object` summaries the basic semantics of `java.lang.Object` from a concurrency point-of-view. And finally, the theory `jmlc` specifies the formal meaning of all model-based concurrency specifications via semantic properties.

To relate the Java, JML and Concurrency theories, we have developed a parametric model-theoretic refinement theory, which is not yet complete. The core goal of the theory is to permit reasoning about arbitrary (semantic) properties at any refinement level, thereby ensuring that properties (such as deadlock-freedom of an *architecture*) are preserved via refinement.

For example, presuming the BON specification of an architecture is deadlock-free, and refinement is formally checkable, then the model-based JML specification of the system is also deadlock-free, and furthermore the Java implementation of that JML specification is deadlock-free. One no longer needs to reason at the code-level about properties that are expressed and reasoned about at the model level. Confirming this hypothesis is the focus of current work.

Chapter 8

Conclusion

We conclude by comparing the described achievements of deliverable D6.3 with its description in the 'Technical Annex' of the CHARTER project:

> D6.3: JML Model for Realtime Concurrency (month 18)
> A set of JML models formalizing key concurrency constructs in SCJ.

Rather than only focusing on native JML types relating to all of the low-level (i.e., locks, blocks, monitors, and threads), and some of the key high-level concurrency constructs available in Real-time Java, we have instead expanded our mandate and included the ability to write annotations on models at arbitrary levels in a refinement sequence via semantic properties.

These native concurrency types (JML and otherwise) are mechanically formally specified in such a way that reasoning about existing current Real-Time Java and Safety-Critical Java code can be performed primarily, and perhaps exclusively, at the model level.

However, we did not accomplish as much as we desired. A formal native model of RTSJ's threading framework is necessary to reason about scheduling and thread control at a model level. A mechanization of the RTSJ memory model may also have utility, as one might wish to talk about different kinds of memory in JML assertions.

The next step to be taken within this line of work in the CHARTER project is the design and development of the *VerCors tool* tool, to be done in the remaining months of Task 6.3:

> D6.4: A Concurrency Checker (month 24)
> A prototype for checking concurrency properties of realtime Java programs.

Appendix A

Histogram Case Study Analysis Summary

What follows are the histograms reported by the CHARTER Histogram tool.

Table A.1: Top 25 classes in Qualitas Corpus by reference count

refs	projs	class name
2633	17	java.util.concurrent.locks.Lock
1848	30	java.util.concurrent.ConcurrentHashMap
1331	15	java.util.concurrent.ConcurrentMap
1330	26	java.util.concurrent.atomic.AtomicInteger
1094	19	java.util.concurrent.atomic.AtomicLong
1051	14	java.util.concurrent.atomic.AtomicBoolean
644	22	java.util.concurrent.locks.ReentrantLock
586	18	java.util.concurrent.locks.ReentrantReadWriteLock
534	13	java.util.concurrent.locks.ReadWriteLock
436	21	java.util.concurrent.CopyOnWriteArrayList
421	20	java.util.concurrent.Future
387	15	java.util.concurrent.CountDownLatch
376	10	java.util.concurrent.ConcurrentLinkedQueue
375	16	java.util.concurrent.ThreadPoolExecutor
331	9	java.util.concurrent.atomic.AtomicReference
321	12	java.util.concurrent.locks.ReentrantReadWriteLock$WriteLock
290	24	java.util.concurrent.ExecutorService
278	13	java.util.concurrent.BlockingQueue
263	10	java.util.concurrent.locks.ReentrantReadWriteLock$ReadLock
213	8	java.util.concurrent.Semaphore
209	22	java.util.concurrent.LinkedBlockingQueue
156	9	java.util.concurrent.locks.Condition
149	10	java.util.concurrent.FutureTask
145	26	java.util.concurrent.Executors
142	14	java.util.concurrent.TimeUnit

Table A.2: Top 25 classes in Qualitas Corpus by project count

refs	projs	class name
1848	30	java.util.concurrent.ConcurrentHashMap
1330	26	java.util.concurrent.atomic.AtomicInteger
145	26	java.util.concurrent.Executors
290	24	java.util.concurrent.ExecutorService
644	22	java.util.concurrent.locks.ReentrantLock
209	22	java.util.concurrent.LinkedBlockingQueue
436	21	java.util.concurrent.CopyOnWriteArrayList
421	20	java.util.concurrent.Future
1094	19	java.util.concurrent.atomic.AtomicLong
586	18	java.util.concurrent.locks.ReentrantReadWriteLock
2633	17	java.util.concurrent.locks.Lock
375	16	java.util.concurrent.ThreadPoolExecutor
1331	15	java.util.concurrent.ConcurrentMap
387	15	java.util.concurrent.CountDownLatch
140	15	java.util.concurrent.Executor
1051	14	java.util.concurrent.atomic.AtomicBoolean
142	14	java.util.concurrent.TimeUnit
534	13	java.util.concurrent.locks.ReadWriteLock
278	13	java.util.concurrent.BlockingQueue
76	13	java.util.concurrent.ExecutionException
321	12	java.util.concurrent.locks.ReentrantReadWriteLock$WriteLock
70	12	java.util.concurrent.CopyOnWriteArraySet
93	11	java.util.concurrent.ScheduledExecutorService
376	10	java.util.concurrent.ConcurrentLinkedQueue
263	10	java.util.concurrent.locks.ReentrantReadWriteLock$ReadLock

Table A.3: Top 25 methods in Qualitas Corpus by reference count

refs	projs	method name
1817	17	java.util.concurrent.locks.Lock.unlock() : void
873	30	java.util.concurrent.ConcurrentHashMap.<init>() : void
724	17	java.util.concurrent.locks.Lock.lock() : void
346	12	java.util.concurrent.atomic.AtomicBoolean.get() : boolean
345	22	java.util.concurrent.atomic.AtomicInteger.get() : int
340	14	java.util.concurrent.atomic.AtomicLong.get() : long
310	13	java.util.concurrent.atomic.AtomicBoolean.set(boolean) : void
292	14	java.util.concurrent.ConcurrentMap.get(java.lang.Object) : java.lang.Object
277	21	java.util.concurrent.atomic.AtomicInteger.<init>(int) : void
270	13	java.util.concurrent.locks.ReadWriteLock.writeLock() : java.util.concurrent.locks.Lock
264	13	java.util.concurrent.locks.ReadWriteLock.readLock() : java.util.concurrent.locks.Lock
245	9	java.util.concurrent.locks.ReentrantReadWriteLock.readLock() : java.util.concurrent.locks.ReentrantReadWriteLock$ReadLock
238	13	java.util.concurrent.ConcurrentHashMap.get(java.lang.Object) : java.lang.Object
231	11	java.util.concurrent.locks.ReentrantReadWriteLock.writeLock() : java.util.concurrent.locks.ReentrantReadWriteLock$WriteLock
227	21	java.util.concurrent.CopyOnWriteArrayList.<init>() : void
222	12	java.util.concurrent.locks.ReentrantLock.unlock() : void
208	12	java.util.concurrent.locks.ReentrantReadWriteLock$WriteLock.unlock() : void
201	19	java.util.concurrent.atomic.AtomicInteger.incrementAndGet() : int
184	15	java.util.concurrent.atomic.AtomicLong.<init>(long) : void
177	12	java.util.concurrent.ConcurrentHashMap.put(java.lang.Object,java.lang.Object) : java.lang.Object
175	22	java.util.concurrent.locks.ReentrantLock.<init>() : void
165	13	java.util.concurrent.atomic.AtomicBoolean.<init>(boolean) : void
165	10	java.util.concurrent.locks.ReentrantReadWriteLock$ReadLock.unlock() : void
157	15	java.util.concurrent.CountDownLatch.countDown() : void
155	9	java.util.concurrent.atomic.AtomicReference.get() : java.lang.Object

Table A.4: Top 25 methods in Qualitas Corpus by project count

refs	projs	method name
873	30	java.util.concurrent.ConcurrentHashMap.<init>() : void
345	22	java.util.concurrent.atomic.AtomicInteger.get() : int
175	22	java.util.concurrent.locks.ReentrantLock.<init>() : void
277	21	java.util.concurrent.atomic.AtomicInteger.<init>(int) : void
227	21	java.util.concurrent.CopyOnWriteArrayList.<init>() : void
201	19	java.util.concurrent.atomic.AtomicInteger.incrementAndGet() : int
56	18	java.util.concurrent.ExecutorService.shutdown() : void
34	18	java.util.concurrent.Executors.newSingleThreadExecutor() : java.util.concurrent.ExecutorService
1817	17	java.util.concurrent.locks.Lock.unlock() : void
724	17	java.util.concurrent.locks.Lock.lock() : void
117	17	java.util.concurrent.ConcurrentHashMap.<init>(int) : void
70	17	java.util.concurrent.locks.ReentrantReadWriteLock.<init>() : void
69	17	java.util.concurrent.LinkedBlockingQueue.<init>() : void
52	17	java.util.concurrent.ExecutorService.shutdownNow() : java.util.List
134	16	java.util.concurrent.Future.get() : java.lang.Object
89	16	java.util.concurrent.atomic.AtomicInteger.<init>() : void
60	16	java.util.concurrent.ExecutorService.submit(java.util.concurrent.Callable) : java.util.concurrent.Future
184	15	java.util.concurrent.atomic.AtomicLong.<init>(long) : void
157	15	java.util.concurrent.CountDownLatch.countDown() : void
140	15	java.util.concurrent.Executor.execute(java.lang.Runnable) : void
96	15	java.util.concurrent.CountDownLatch.<init>(int) : void
76	15	java.util.concurrent.atomic.AtomicInteger.getAndIncrement() : int
52	15	java.util.concurrent.Future.get(long,java.util.concurrent.TimeUnit) : java.lang.Object
37	15	java.util.concurrent.ExecutorService.awaitTermination(long,java.util.concurrent.TimeUnit) : boolean
340	14	java.util.concurrent.atomic.AtomicLong.get() : long

Appendix B

HOL Specifications of JML Native Types

What follows are snapshots of the current PVS specifications of the theories summarized in section 7.3.

```
java: THEORY
BEGIN
  IMPORTING strings

  nominal_modules: TYPE+ = [# name: string #]
  classes: TYPE = nominal_modules WITH [# type_parameters: set[nominal_modules],
                                          abstract?: boolean,
                                          final?: boolean #]
  interfaces: TYPE = nominal_modules WITH [# type_parameters: set[nominal_modules] #]

  java_types: NONEMPTY_TYPE
  reference_types: TYPE+ FROM java_types
  primitive_types: TYPE+ = { byte, short, int, long, float, double, boolean }

  exceptions: TYPE+ FROM classes

  variables: TYPE+ = [# java_type: java_types, name: string #]
  visibility_modifiers: TYPE+ = { public, protected, private, package }
  java_field_modifiers: TYPE+ = { static, final, transient, volatile }
  % @todo jrk Need we model abstract classes and methods? - 1 June 2010
  % class_or_method_modifiers: TYPE+ = { abstract }
  formal_parameter: TYPE+ = [# final?: boolean, definition: variables #]

  fields: TYPE+ = [# visibility: visibility_modifiers,
                     java_modifiers: set[java_field_modifiers],
                     definition: variables,
                     class: nominal_modules #]
  formal_parameters: TYPE+ = list[formal_parameter]
  signature: TYPE = [# name: string, arguments: formal_parameters #]
  methods: TYPE = [# visibility: visibility_modifiers,
                     final?: boolean,
                     static?: boolean,
                     abstract?: boolean,
                     return_type: java_types,
                     sig: signature,
                     module: nominal_modules,
                     % overrides is bottom if this method is non-overriding
                     overrides: lift[nominal_modules],
                     implements: set[interfaces],
                     throws: set[exceptions] #]

  m: VAR methods
  % if module is an interface the implements must be empty.
  method_wellformedness: AXIOM (FORALL(m: methods):
    (EXISTS(i: interfaces): m`module`name = i`name) IMPLIES
                                                    empty?(m`implements))

  fields: [classes -> set[fields]]
  methods: [classes -> set[methods]]
  fields: [interfaces -> set[fields]]
  methods: [interfaces -> set[methods]]

  inputs: TYPE+
  outputs: TYPE+

END java
```

Figure B.1: The core Java PVS theory.

```
java_inheritance_semantics [T: TYPE]: THEORY
BEGIN
  IMPORTING java, relations[T]

  t, u: T

  extends: [T -> set[T]]
  inherit_from: [T, T -> bool]

  inherit_from_semantics: AXIOM reflexive?(inherit_from) AND
                                antisymmetric?(inherit_from) AND
                                transitive?(inherit_from)
  extends_induces_inherit_from: AXIOM extends(t) = u => inherit_from(t, u)

  extends_semantics: AXIOM empty?(extends(t)) OR singleton?(extends(t))

END java_inheritance_semantics
```

Figure B.2: The inheritance semantics of Java.

```
java_semantics: THEORY
BEGIN
  IMPORTING java, java_inheritance_semantics[classes],
            java_inheritance_semantics[interfaces]

  c: classes
  i, j: interfaces
  s: set[interfaces]

  implements: [classes -> set[interfaces]]
  implements_is_the_transitive_closure_of_extends_on_interfaces: AXIOM
    implements(c) = s AND member(i, s) AND extends(i) = j =>
      NOT empty?(j) IMPLIES member(epsilon(j), s)

  java_lang_Object: classes = (# name := "java.lang.Object",
                                  type_parameters := emptyset,
                                  abstract? := false, final? := false #)
  classes_extends_codomain_is_cardinality_zero_or_one: AXIOM Card(extends(c)) <= 1
  only_object_extends_nothing: AXIOM extends(c) = emptyset <=> c = java_lang_Object

  null: reference_types

  java_lang_Object_is_the_top_class: AXIOM inherit_from(c, java_lang_Object)

  java_lang_Throwable, java_lang_Error, java_lang_Exception: classes

  Throwable_extends_Object: AXIOM extends(java_lang_Throwable) = java_lang_Object
  Error_extends_Throwable: AXIOM extends(java_lang_Error) = java_lang_Throwable
  Exception_extends_Throwable: AXIOM extends(java_lang_Exception) = java_lang_Throwable

  java_lang_Runnable: interfaces = (# name := "java.lang.Runnable",
                                      type_parameters := emptyset #)
  java_lang_Thread: classes = (# name := "java.lang.Thread",
                                  type_parameters := emptyset,
                                  abstract? := false, final? := false #)
  Thread_extends_Object: AXIOM extends(java_lang_Thread) = java_lang_Object

  all_interface_fields_are_public: AXIOM
    (FORALL (i: interfaces):
       (FORALL (f: fields): member(f, fields(i)) =>
         f`visibility = public))
  all_interface_fields_are_static_final: AXIOM
    (FORALL (i: interfaces):
       (FORALL (f: fields): member(f, fields(i)) =>
         member(final, f`java_modifiers) AND member(static, f`java_modifiers)))
  all_interface_methods_are_public: AXIOM
    (FORALL (i: interfaces):
       (FORALL (m: methods): member(m, methods(i)) =>
         m`visibility = public))

END java_semantics
```

Figure B.3: The core Java semantics necessary for modeling concurrency.

```
jml: THEORY
BEGIN
  IMPORTING java_semantics

  preconditions: TYPE+ = bool
  minmax_frames: TYPE+ = { everything, nothing }
  frame_axioms: TYPE+ = [set[fields] + minmax_frames]
  normal_postconditions: TYPE+ = bool
  exceptional_postcondition: TYPE+ = [# exception: exceptions, post: bool #]

  invariants: TYPE+ = [# visibility: visibility_modifiers, assertion: bool #]
  constraints: TYPE+ = [# visibility: visibility_modifiers, assertion: bool #]
  initially_clauses: TYPE+ = [# visibility: visibility_modifiers, assertion: bool #]

  jml_method_modifiers: TYPE+ = { pure }
  jml_methods: TYPE = methods WITH [# jml_modifiers: set[jml_method_modifiers] #]

  ghost_fields: TYPE+ = set[fields]
  model_fields: TYPE+ = set[fields]
  model_methods: TYPE+ = set[methods]

  datagroups: TYPE+ = [ set[fields], set[model_fields] ]
  represents: [fields, model_fields -> bool]
  nullable?: [reference_types -> bool]

  contracts: TYPE+ = [# visibility: visibility_modifiers,
                        pre: preconditions,
                        frame: frame_axioms,
                        post: normal_postconditions,
                        ex_post: set[exceptional_postcondition] #]
  normal_behaviors: TYPE+ = { c: contracts | empty?(c`ex_post) }
  exceptional_behaviors: TYPE+ = { c: contracts | empty?(c`post) }
  behaviors: TYPE+ = contracts

  contract: [jml_methods -> contracts]
  invariant: [nominal_modules -> set[invariants]]
  contraint: [nominal_modules -> set[constraints]]
  initially: [nominal_modules -> set[initially_clauses]]

  model_fields_of_class: [classes -> set[model_fields]]
  model_methods_of_class: [classes -> set[model_methods]]
  ghost_fields_of_class: [classes -> set[ghost_fields]]
  model_fields_of_interface: [interfaces -> set[model_fields]]
  model_methods_of_interface: [interfaces -> set[model_methods]]
  ghost_fields_of_interface: [interfaces -> set[ghost_fields]]

  pure_means_modifies_nothing: AXIOM
    (FORALL (m: jml_methods): (m`jml_modifiers = pure) =>
    (OUT_2(contract(m)`frame) = nothing))

END jml
```

Figure B.4: The core JML notions.

```
Concurrency : DATATYPE WITH SUBTYPES concurrent, guarded, failure %, sequential
BEGIN
  IMPORTING java

  %sequentual(desc:string): sequential? : sequential %Sequential is just concurrent 1
  concurrent(bound:nat, desc:string): concurrent? : concurrent
  guarded_lock(lock:string, desc:string): guarded_lock? : guarded
  guarded_bound(bound:nat, desc:string): guarded_bound? : guarded
  failure(bound:nat, exception:classes, desc:string): failure? : failure
END Concurrency

concurrency : THEORY
BEGIN
  IMPORTING Concurrency

  guarded_lock: TYPE = (guarded_lock?)
  guarded_bound: TYPE = (guarded_bound?)
END concurrency
```

Figure B.5: The concurrency specification constructs.

```
semantic_property [T:TYPE] : THEORY
BEGIN
  IMPORTING java, jml

  %SemanticProperty: TYPE
  %get_prop: [SemanticProperty -> T]
  props_of_class: [classes -> set[T]]
  props_of_field: [fields -> set[T]]
  props_of_method: [methods -> set[T]]
  props_of_interface: [interfaces -> set[T]]
END semantic_property

defaults : THEORY
BEGIN
  IMPORTING concurrency, max_nat

  DefaultConcurrent: set[concurrent] = singleton(concurrent(infinity, ""))
  DefaultGuardedLock: set[guarded_lock] = emptyset
  DefaultGuardedBound: set[guarded_bound] = emptyset
  DefaultFailure: set[failure] = emptyset

  DefaultConcurrency: set[Concurrency] =
    union[Concurrency](DefaultConcurrent,
    union[Concurrency](DefaultGuardedLock,
    union[Concurrency](DefaultGuardedBound, DefaultFailure)))

END defaults

object : THEORY
BEGIN
  IMPORTING java_semantics, defaults, semantic_property[Concurrency]

  c: VAR classes

  Object: classes
  object_is_top_type: AXIOM (FORALL c: c=Object OR inherit_from(c,Object))
  object_methods_all_default: AXIOM (FORALL (m:(methods(Object))):
    props_of_method(m) = DefaultConcurrency)
END object
```

Figure B.6: The structure and semantics of semantic properties.

```
subtype_set[T:TYPE, U: TYPE FROM T] : THEORY
BEGIN
  S: VAR set[T]

  subtype_set(S): set[U] = { s:U | S(s) }
%  x: set[U FROM T]
END subtype_set

java_additions : THEORY
BEGIN
  IMPORTING java_semantics, object

  c1,c2: VAR classes

  inherit_from(c1)(c2): bool = inherit_from(c1,c2)

END java_additions

max_nat: THEORY
BEGIN
  x, y: VAR nat
  maximum?(x): bool = (FORALL y: y <= x)
  infinity: (maximum?)
END max_nat

set_stuff [T: TYPE]: THEORY
BEGIN
  IMPORTING sets[T]
  p: VAR PRED[T]
  S: VAR set[T]
  t: VAR T
  none(p)(S): bool  = FORALL (s:(S)): NOT(p(s))
  none_is_not_some: LEMMA none(p)(S) IFF NOT(some(p)(S))
  none_is_not_exists: LEMMA none(p)(S) IFF NOT(EXISTS (x:(S)): p(x))
END set_stuff
```

Figure B.7: Incidental supporting theories.

```
jmlc : THEORY
BEGIN
  IMPORTING concurrency, semantic_property[Concurrency], java_semantics, java_additions

  s: VAR set[Concurrency]
  m: VAR methods
  c,c1,c2: VAR classes

  %Helper extraction functions
  guarded_bounds(s): set[guarded_bound] = { p:(guarded_bound?) | s(p) }
  guarded_bounds(m): set[guarded_bound] = guarded_bounds(props_of_method(m))
  guarded_bounds(c): set[guarded_bound] = guarded_bounds(props_of_class(c))
  guarded_locks(s): set[guarded_lock] = { p:(guarded_lock?) | s(p) }
  guarded_locks(m): set[guarded_lock] = guarded_locks(props_of_method(m))
  guarded_locks(c): set[guarded_lock] = guarded_locks(props_of_class(c))
  concurrent(s): set[concurrent] = { p:concurrent | s(p) }
  concurrent(m): set[concurrent] = concurrent(props_of_method(m))
  concurrent(c): set[concurrent] = concurrent(props_of_class(c))
  failure(s): set[failure] = { p:failure | s(p) }
  failure(m): set[failure] = failure(props_of_method(m))
  failure(c): set[failure] = failure(props_of_class(c))
  locks_for_method(m): set[string] = { s:string | (EXISTS (l:(guarded_locks(m))): lock(l)=s) }
  locks_for_class(c): set[string] = { s:string |  (EXISTS (l:(guarded_locks(c))): lock(l)=s) }

  parent_method_im_overriding: [methods -> lift[methods]] %TODO move out of jmlc
  no_parent_method_im_overriding?(m): bool = bottom?(parent_method_im_overriding(m))
  top_class?(c): bool = c=Object
  overridden_method_properties(m): set[Concurrency] =
    CASES parent_method_im_overriding(m) OF
      bottom: emptyset,
      up(parent_method): props_of_method(parent_method)
    ENDCASES

  implemented_interface_methods_im_implementing: [methods -> set[methods]]

  overridden_and_implemented_methods(m): set[methods] =
    LET interface_methods:set[methods]=implemented_interface_methods_im_implementing(m) IN
    CASES parent_method_im_overriding(m) OF
      bottom: interface_methods,
      up(parent_method): union(interface_methods, parent_method)
    ENDCASES

  %TODO validity of some collection of java structures, not universal validity!

  child_gbs, parent_gbs: VAR set[guarded_bound]

  valid_guarded_bound_inheritance(child_gbs, parent_gbs): bool =
    (empty?(child_gbs) AND empty?(parent_gbs))
   OR
    (singleton?(child_gbs) AND singleton?(parent_gbs) AND bound(parent_gbs) <= bound(child_gbs))
```

Figure B.8: The semantics of concurrent semantic properties (part 1).

```
%For every method, either there is no guarded bound or there is one guarded bound.
%If there is a guarded bound, and this method overrides a parent method, then that
%parent method must have a guarded bound less than or equal to this bound.
valid_method_guarded(m): bool =
  no_parent_method_im_overriding?(m) OR
  FORALL (pm:(overridden_and_implemented_methods(m))):
    valid_guarded_bound_inheritance(guarded_bounds(m), guarded_bounds(pm))

valid_class_guarded(c): bool =
  top_class?(c) OR
  (FORALL (p:(inherit_from(c))):
    valid_guarded_bound_inheritance(guarded_bounds(c), guarded_bounds(p)))

%Every method must require the same locks as the parent method it overrides (if it does override).
valid_method_locks(m): bool =
  no_parent_method_im_overriding?(m) OR
  FORALL (pm:(overridden_and_implemented_methods(m))):
    locks_for_method(m)=locks_for_method(pm)

valid_class_locks(c): bool =
  top_class?(c) OR (FORALL (p:(inherit_from(c))): locks_for_class(c) = locks_for_class(p) )

child_cs, parent_cs: VAR set[concurrent]
valid_concurrent_inheritance(child_cs, parent_cs): bool =
  (empty?(child_cs) AND empty?(parent_cs))
 OR
  (singleton?(child_cs) AND singleton?(parent_cs) AND bound(parent_cs) <= bound(child_cs))

%For every method, either there is no concurrency bound or there is one concurrency bound.
%If there is a concurrency bound, and this method overrides a parent method, then that
%parent method must have a concurrency bound less than or equal to this bound.
valid_method_concurrent(m): bool =
  no_parent_method_im_overriding?(m) OR
  valid_concurrent_inheritance(concurrent(m), concurrent(overridden_method_properties(m)))

valid_class_concurrency?(c): bool =
  top_class?(c) OR (FORALL (p:(inherit_from(c))):
    valid_concurrent_inheritance(concurrent(c), concurrent(p)))
```

Figure B.9: The semantics of concurrent semantic properties (part 2).

```
  %FAILURE
  %Each child and class has at most one failure property.
  %In the prescence of inheritance either the method/class has no
  %failure clause and the parent

  child_fs, parent_fs: VAR set[failure]
  valid_failure_inheritance?(child_fs, parent_fs): bool =
    (empty?(child_fs) AND empty?(parent_fs))
   OR
    (    singleton?(child_fs)
     AND singleton?(parent_fs)
     AND bound(child_fs) = bound(parent_fs)
     AND inherit_from(exception(child_fs), exception(parent_fs)))


  valid_method_failure(m): bool =
    no_parent_method_im_overriding?(m) OR
    valid_failure_inheritance?(failure(m), failure(overridden_method_properties(m)))

  valid_class_failure?(c): bool =
    top_class?(c) OR (FORALL (p:(inherit_from(c))): valid_failure_inheritance?(failure(c), failure(p)))

  %TODO need a supertype of classes and interfaces that has methods defined on
  %valid class w.r.t. method (all method props must be identical).
  valid_method_class: bool =
    (FORALL c:
      (FORALL (m:(methods(c))): props_of_method(m)=props_of_class(c)))
END jmlc
```

Figure B.10: The semantics of concurrent semantic properties (part 3).

```
refinement[T, U: TYPE,
           (IMPORTING relations[T]) ==: equivalence,
           (IMPORTING relations[U]) ##: relations[U].equivalence,
           gen: [T -> U],
           ext: [U -> T],
           |-: predicate[[T, U]]]: THEORY
BEGIN
  ASSUMING
    t, t0, t1: VAR T
    u, u0, u1: VAR U

    generate_consistent_with_refinement: ASSUMPTION t |- gen(t)

    extract_consistent_with_refinement: ASSUMPTION ext(u) |- u

    ext_is_left_inv_of_gen: ASSUMPTION ext(gen(t)) == t

    gen_is_left_inv_of_ext: ASSUMPTION gen(ext(u)) ## u

    gen_preserves_equiv: ASSUMPTION u0 = gen(t0) AND u1 = gen(t1) AND
                                    t0 == t1 IMPLIES u0 ## u1

    ext_preserves_equiv: ASSUMPTION t0 = ext(u0) AND t1 = ext(u1) AND
                                    u0 ## u1 IMPLIES t0 == t1
 ENDASSUMING

  equal_ext_to_equality: LEMMA FORALL((t:T),(u:U)):
                               u = gen(t) AND t = ext(u)
                               IMPLIES ext(u)== ext(gen(t)) %-:)

  ref_extract_generate_equal: LEMMA FORALL((t:T),(u:U)):
                                    u ## gen(t) AND t == ext(u)
                                    IMPLIES ext(u) == ext(gen(t)) %-:)

  equal_gen_to_equality: LEMMA FORALL((t:T),(u:U)):
                               u = gen(t) AND t= ext(u)
                               IMPLIES gen(t)## gen(ext(u)) %-:)

  ref_generate_extract_equal: LEMMA FORALL((t:T),(u:U)):
                                    u ## gen(t) AND t == ext(u)
                                    IMPLIES gen(t) ## gen(ext(u)) %-:)

  ref_left_composition: LEMMA FORALL ((t:T),(u:U)):
                              u = gen(t) AND |-(t,u) IMPLIES |-(ext(gen(t)),u) %-:) ?

  ref_right_composition: LEMMA FORALL ((t:T),(u:U)):
                               t = ext(u) AND |-(t,u) IMPLIES |-(t,gen(ext(u))) %-:) ?

  ref_composition: LEMMA FORALL ((t:T),(u:U)):
                         u = gen(t) AND t = ext(u) AND
                         |-(t,u) IMPLIES |-(ext(gen(t)),gen(ext(u))) %-:) ?
```

Figure B.11: A higher-order theory of refinement (part 1).

```
  equivalence_ref_impl: LEMMA FORALL(t0:T),(u0:U):
                              u0 ## gen(t0) AND
                              t0 == ext(u0) AND
                              |-(t0,u0) IMPLIES |-(ext(u0),gen(t0))

  rep_equiv_ref_impl: LEMMA FORALL(t0:T),(u0:U):
                            u0 ## QuotientDefinition[U].repEC( ## )(gen(t0)) AND
                            t0 == QuotientDefinition[T].repEC(==)(ext(u0)) AND
                            |-(t0,u0) IMPLIES |-(ext(u0),gen(t0))

  rep_equiv_ext_ref_impl: LEMMA FORALL(t0:T),(u0:U):
                                u0 ## QuotientDefinition[U].repEC( ## )(gen(t0)) AND
                                t0 == QuotientDefinition[T].repEC(==)(ext(u0)) AND
                                |-(t0,u0) IMPLIES |-(QuotientDefinition[T].repEC(==)(ext(u0)),
                                                    QuotientDefinition[U].repEC( ## )(gen(t0)))
END refinement
```

Figure B.12: A higher-order theory of refinement (part 2).

```
refinement_model_theory[T, U: TYPE,
                        (IMPORTING relations[T]) ==: equivalence,
                        (IMPORTING relations[U]) ==: relations[U].equivalence,
                        gen: [T -> U],
                        ext: [U -> T],
                        |-: predicate[[T, U]],
                        [||]: [predicate[T] -> predicate[U]],
                        |=: predicate[[T, predicate[T]]],
                        |=: predicate[[U, predicate[U]]]]: THEORY
BEGIN
  ASSUMING
  IMPORTING refinement[T, U, ==, ==, gen, ext, |-]

    p: VAR predicate[T]
    q: VAR predicate[U]
    t: VAR T
    u: VAR U

    interpretation_preserves_validity: ASSUMPTION (t |= p) IMPLIES (u |= [|p|])
  ENDASSUMING
END refinement_model_theory
```

Figure B.13: The signature of a model-theoretic refinement theory.

Bibliography

[1] M. Abadi, C. Flanagan, and S. Freund. Types for safe locking: Static race detection for java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2), 2006.

[2] M. Abadi, C. Flanagan, and S. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255, 2006.

[3] S. Adve and H. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8), 2010.

[4] W. Araujo, L. Briand, and Y. Labiche. Enabling the runtime assertion checking of concurrent contracts for the java modeling language. In *Proceedings of ICSE'11*. IEEE, 2011.

[5] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison–Wesley Publishing Company, fourth edition, Aug. 2005.

[6] N. Ayewah, W. Pugh, J. Morgenthaler, J. Penix, and Y. Zhou. Using FindBugs on production software. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*. ACM Press, 2007.

[7] B. Beckert and V. Klebanov. A dynamic logic for deductive verification of concurrent programs. In *Proceedings of SEFM'07*. IEEE, 2007.

[8] J. Bengtson, J. Jensen, F. Sieczkowski, and L. Birkedal. Verifying object-oriented programs with higher-order separation logic in coq. *Proceedings of ITP'11*, 2011.

[9] S. Blom and J. R. Kiniry. The histogram tool set, 2011.

[10] H. Boehm and S. Adve. You don't know jack about shared variables or memory models. *Communications of the ACM*, 55(2), 2012.

[11] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Turnbull, and R. Belliardi. *The Real-Time Specification for Java*. Oracle America, Inc., second edition, 2005. JSR-000001.

[12] S. Boroday, A. Petrenko, J. Singh, and H. Hallal. Dynamic analysis of java applications for multithreaded antipatterns. *ACM SIGSOFT Software Engineering Notes*, 30(4), 2005.

[13] G. Boudol and G. Petri. Relaxed memory models: an operational approach. *Proceedings of POPL'09*, 44(1), 2009.

[14] K. P. Boysen and G. T. Leavens. Discussion of design alternatives for JML Java 5 annotations. Technical Report 08-01, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, Jan. 2008.

[15] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, Feb. 2005.

[16] O. Burn. Checkstyle homepage, 2012.

[17] T. Cargill. *Proceedings of International Conference on Pattern Languages of Programming (PLoP '96)*, chapter Specific Notification for Java Thread Synchronization. Number wucs-97-07 in Washington University Technical Report Series. Washington University, 1997.

[18] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Proceedings of the International Symposium on Formal Methods for Components and Objects (FMCO)*, volume 4111 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[19] K. Chandy. Concurent program archetypes. In *Proceedings of the 1994 IEEE Scalable Parallel Libraries Conference*, 1994.

[20] Contracts for java. http://code.google.com/p/cofoja/, Nov. 2011.

[21] D. R. Cok and J. R. Kiniry. The Extended Static Checker for Java, version 2, 2003–. See http://kindsoftware.com/products/opensource/ESCJava2/.

[22] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*. ACM Press, 2000.

[23] A. De, A. Roychoudhury, and D. D'Souza. Java memory model aware sofware verification. Technical report, Technical report, 2008. http://clweb.csa.iisc.ernet.in/arnabde/opmm-full.pdf, 2010.

[24] X. Deng, M. B. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, New York, NY, USA, 2002. Association of Computing Machinery.

[25] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3), Dec. 2007.

[26] F. Fairmichael and J. Kiniry. The BONc tool. Details available via http://kindsoftware.com/products/opensource/BONc/, 2010.

[27] Findbugs. http://findbugs.sourceforge.net/, 2012.

[28] C. Flanagan and S. Freund. Type-based race detection for java. In *Proceedings of PLDI'00*. ACM, 2000.

[29] C. Flanagan and S. Freund. Detecting race conditions in large programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2001.

[30] C. Flanagan and S. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *Proceedings of POPL'04*, 39(1), 2004.

[31] C. Flanagan and S. Freund. Fasttrack: efficient and precise dynamic race detection. In *Proceedings of PLDI'09*. ACM, 2009.

[32] C. Flanagan, S. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. *Programming Languages and Systems*, 2305, 2002.

[33] C. Flanagan, S. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of PLDI'08*. ACM, 2008.

[34] C. Flanagan, S. N. Freund, M. Lifshin, and S. Quadeer. Types for atomicity: Static checking and inference for java. *ACM Transactions on Programming Languages and Systems*, 30(4), July 2008.

[35] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. *ACM SIGPLAN Notices*, 37(5), 2002.

[36] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison–Wesley Publishing Company, 2006.

[37] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison–Wesley Publishing Company, first edition, Aug. 1996.

[38] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison–Wesley Publishing Company, second edition, June 2000.

[39] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison–Wesley Publishing Company, third edition, June 2005.

[40] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification*. Oracle America, Inc., java se 7 edition edition, July 2011. PDF edition dated 2012-02-06.

[41] D. Grossman, J. Manson, and W. Pugh. What do high-level memory models mean for transactions? In *Proceedings of the 2006 workshop on Memory system performance and correctness*. ACM, 2006.

[42] J. Guttag, J. J. Horning, et al., editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[43] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's reentrant locks (or Separation logic for Java's reentrant locks!). In G. Ramalingam, editor, *Asian Symposium on Programming Languages and Systems (APLAS'08)*, volume 5356 of *Lecture Notes in Computer Science*. Springer-Verlag, Dec. 2008.

[44] P. B. Hansen. Java's insecure parallelism. Technical report, L.C. Smith College of Engineering and Computer Science - Former Departments, Centers, Institutes and Projects, 1999. Paper 11.

[45] J. Hatcliff and M. Dwyer. Using the bandera toolset to model-check properties of concurrent java software. *Proceedings of CONCUR'01*, 2001.

[46] M. Huisman and C. Hurlin. The stability problem for verification of concurrent object-oriented programs. In *Proceedings of Verification and Analysis of Multi-threaded Java-like Programs (VAMP'07)*, Sept. 2007.

[47] C. Hurlin. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic*. PhD thesis, Université Nice - Sophia Antipolis, Sept. 2009.

[48] C. Hurlin. Specifying and checking protocols of multithreaded classes. In *ACM Symposium on Applied Computing (SAC'09), Software Verification and Testing Track*. Association of Computing Machinery, Mar. 2009.

[49] B. Jacobs. *A Statically Verifiable Programming Model for Concurrent Object-Oriented Programs*. PhD thesis, Katholieke Universiteit Leuven, 2007.

[50] B. Jacobs and E. Poll. Java program verification at nijmegen: Developments and perspective. *Software Security-Theories and Systems*, 2004.

[51] B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *Proceedings of the Eighth International Conference on Formal Engineering Methods (ICFEM 2006)*, volume 4260 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[52] B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A simple sequential reasoning approach for sound modular verification of mainstream multithreaded programs. *Electronic Notes in Theoretical Computer Science*, 174(9), 2007. Special issue on Thread Verification (TV'06).

[53] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):21, 2009.

[54] H. Jin, T. Yavuz-Kahveci, and B. Sanders. Java memory model-aware model checking. *Department of Computer and Information Science, University of Florida, Tech. Rep. REP-2011-516*, 2011.

[55] JSR 305 Expert Group. Annotations for software defect detection. Java specification request, Java Community Process, 2007. See also http://groups.google.com/group/jsr-305.

[56] JSR 308 Expert Group. Annotations on Java types. Java specification request, Java Community Process, 2007. See also http://types.cs.washington.edu/jsr308.

[57] The JSR-302 Home Page: Safety Critical Javatm Technology. http://www.jcp.org/en/jsr/-detail?id=302.

[58] J. R. Kiniry. *Kind Theory*. PhD thesis, Department of Computer Science, California Institute of Technology, 2002.

[59] J. R. Kiniry and D. R. Cok. ESC/Java2: Uniting ESC/Java and JML. In *Proceeding of the International Workshop on the Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS) 2004*, volume 3362 of *Lecture Notes in Computer Science*. Springer-Verlag, Jan. 2005.

[60] J. R. Kiniry and F. Fairmichael. Ensuring consistency between designs, documentation, formal specifications, and implementations. In *Proceedings of the 12th International Symposium on Component Based Software Engineering ("CBSE" '09)*, volume 5582 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[61] J. R. Kiniry and D. M. Zimmerman. The Infospheres Java coding standard. Technical report, Department of Computer Science, California Institute of Technology, 1997.

[62] KSU SAVES project. Syncgen.
Available via http://syncgen.projects.cis.ksu.edu/, 2004.

[63] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison–Wesley Publishing Company, second edition, 1999.

[64] G. Leavens et al. The common jml tools. http://www.jmlspecs.org/, 2012.

[65] G. Leavens et al. The java modeling language (jml). http://www.jmlspecs.org/, July 2012.

[66] G. T. Leavens, A. L. Baker, and C. Ruby. *Behavioral Specifications of Business and Systems*, chapter JML: A Notation for Detailed Design. Kluwer Academic Publishing, 1999.

[67] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual. Draft Revision 1.200*, Feb. 2007.

[68] J. Lee, Robby, and P. Chalin. The java contract projects. http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/JavaContract/, 2009.

[69] R. A. Lerner. *Specifying objects of concurrent systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.

[70] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification*. Oracle America, Inc., java se 7 edition edition, June 2011. PDF edition dated 2012-02-06.

[71] P. Louridas. Static code analysis. *IEEE Software*, 23(4), 2006.

[72] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. pub-wiley, 2006.

[73] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *Proceedings of POPL'05*. ACM Press, 2005.

[74] A. Martin and J. Burch. Fair mutual exclusion with unfair P and V operations. *Information Processing Letters*, 21(2), 1985.

[75] V. Menon, S. Balensiefer, T. Shpeisman, A. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for java stm. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. ACM, 2008.

[76] M. Mizuno. A structured approach for developing concurrent programs in java. *Information Processing Letters*, 69(5), 1999.

[77] M. Mizuno, G. Singh, and M. Neilsen. A structured approach to develop concurrent programs in UML. In *Proceedings of the Third International Conference on the Unified Modeling Language*, volume 1939 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[78] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of OSDI'08*. USENIX Association, 2008.

[79] P. Nienaltowski, B. Meyer, and J. S. Ostroff. Contracts for concurrency. *Formal Aspects of Computing*, 21(4), Aug. 2009.

[80] S. Oaks and H. Wong. *Java Threads*. O'Reilly & Associates, Inc., 2004.

[81] Parasoft. Using design by contract to automate software and component testing. Technical report, Parasoft, Inc., 2002. last visited: July 2012.

[82] Pmd. http://pmd.sourceforge.net/, 2012.

[83] E. Rodríguez, M. Dwyer, C. Flanagan, J. Hatcliff, and G. T. Leavens. Extending JML for modular specification and verification of multi-threaded programs. In *ECOOP, LNCS 3586*. Springer-Verlag, 2005.

[84] A. Roscoe. Model-checking csp. *A Classical Mind, Essays in Honour of CAR Hoare*, 1994.

[85] C. Sadowski, T. Ball, J. Bishop, S. Burckhardt, G. Gopalakrishnan, J. Mayo, M. Musuvathi, S. Qadeer, and S. Toub. Practical parallel and concurrent programming. In *Proceedings SIGCSE'11*. ACM, 2011.

[86] R. SC-167 and E. WG-12. Do-178b, software considerations in airborne systems and equipment certification. Technical report, RTCA, Incorporated, 1992.

[87] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010.

[88] H. Thimbleby. A critique of java. *Software-Practice and Experience*, 29(5), 1999.

[89] K. Trentelman and M. Huisman. Extending JML specifications with temporal logic. *Algebraic Methodology and Software Technology*, 2002.

[90] K. Waldén and J.-M. Nerson. *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems*. Prentice–Hall, Inc., 1995.

[91] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison–Wesley Publishing Company, 1998.

[92] A. Wellings. *Concurrent and Real-Time Programming in Java*. John Wiley and Sons Ltd, 2004.

[93] T. Wilson, S. Maharaj, and R. Clark. Omnibus verification policies: A flexible, configurable approach to assertion-based software verification. In *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*. IEEE Press, 2005.