Extending JML for Modular Specification and Verification of Multi-threaded Programs

Edwin Rodríguez[1], Matthew Dwyer[2], Cormac Flanagan[3], John Hatcliff[1], Gary T. Leavens[4], and Robby[1]

[1] Department of Computing and Information Sciences, Kansas State University
{edwin, hatcliff, robby}@cis.ksu.edu

[2] Department of Computer Science and Engineering, University of Nebraska-Lincoln
dwyer@cse.unl.edu

[3] Computer Science Department, University of California at Santa Cruz
cormac@cs.ucsc.edu

[4] Department of Computer Science, Iowa State University
leavens@cs.iastate.edu

Abstract. The Java Modeling Language (JML) is a formal specification language for Java that allows developers to specify rich software contracts for interfaces and classes, using pre- and postconditions and invariants. Although JML has been widely studied and has robust tool support based on a variety of automated verification technologies, it shares a problem with many similar object-oriented specification languages—it currently only deals with sequential programs. In this paper, we extend JML to allow for effective specification of multi-threaded Java programs. The new constructs rely on the non-interference notion of *method atomicity*, and allow developers to specify locking and other non-interference properties of methods. Atomicity enables effective specification of method pre- and postconditions and supports Hoare-style modular reasoning about methods. Thus the new constructs mesh well with JML's existing features. We validate the specification language design by specifying the behavior of a number of complex Java classes designed for use in multi-threaded programs. We also demonstrate that it is amenable to automated verification using model checking technology.

1 Introduction

The use of rich source-level specification languages for expressing correctness properties of object-oriented programs is growing in practice. Specification languages such as the Java Modeling Language (JML) [1, 2, 3, 4] and Spec# [5] provide a wide range of light-weight annotations (e.g., specifying non-nullness of variables of reference type) as well as constructs for writing specifications of full functional behaviors of class implementations that can be checked by a variety of verification technologies including static analysis, run-time monitoring, model checking, and theorem-proving. JML is a behavioral interface specification language that allows developers to specify both the syntactic and behavioral interface of a portion of Java code. It supports the design by contract paradigm [6] by including notation for pre- and postconditions and invariants. JML uses Java's expression syntax and adds features for: universal (`\forall`) and existential (`\exists`) quantification over object instances as well as basic types, such as integers, and constructs for expressing properties of heap allocated data (such as `\reach` which returns the set of objects reachable from a particular reference).

JML has proved to be an effective vehicle for bringing together a number of research teams [1] seeking to (a) extend the logical foundations of specification formalisms needed for addressing semantically complex language features such as dynamic dispatch, exceptions, dynamic object creation, and (b) build tool support for automated and computer-assisted reasoning about real-world Java applications. However, despite the success of JML in specifying programs written in sequential Java and its Java Card dialect, JML's support for concurrency "is still in its infancy" [4].

Although many interesting programs are sequential, the flexibility that accompanies concurrent programming in terms of further modularizing the design (thread modularity), means that most moderately complex systems are programmed with some sort of concurrency modality (multi-threading, multi-processing, etc.). Moreover, multi-threading capabilities are becoming more accessible to programmers since languages like Java and C# provide direct language support for threads, while other languages provide sophisticated support via libraries (e.g., POSIX threads).

Most existing specification and checking tools for multi-threaded programs focus on properties such as absence of race conditions, establishing mutual exclusion, and simple event ordering and temporal properties (e.g., capturing proper ordering of calls to APIs). However, they typically ignore strong functional properties and complex data structure invariants. These inadequacies stem from the challenges of dealing with thread interference in the manipulation of shared (heap) data. One cannot simply apply Hoare-style logics using method pre- and postconditions to reason modularly, because method execution is often *non-serial*—the actions of other threads may interfere with the thread executing the method and thus render invalid assumptions captured in method preconditions and guarantees captured in postconditions.

Due to the pervasiveness of multi-threading and the increasing use of multi-threaded object-oriented code in embedded and mission- and safety-critical applications, it is necessary to extend sequential specification languages like JML to support specification and reasoning about multi-threaded programs. Furthermore, these extensions should allow both light-weight annotations and more complete, functional specifications, and should work with multiple different reasoning tools. This paper advances toward these goals by making the following contributions:

– We identify situations in which the current JML fails to enable effective specification and modular reasoning for multi-threaded programs.
– We identify specification forms that we and other researchers have found useful for multi-threaded programs. This includes (a) various light-weight

annotations that can be leveraged by automated checking technologies and (b) the use of atomicity specifications to achieve modular reasoning about methods.
- We show how to integrate these forms into JML in a way that enables both reasoning about data values and concurrency concerns.
- We validate the design of this enhanced version of JML by using it to specify properties of a number of Java libraries designed for concurrency, including most of the concurrent data structure classes from `java.util.concurrent`, which includes some very intricate concurrent Java code (the full collection of these specified examples are posted on our project web-site [7]).
- We establish that these specification formalisms are amenable to effective automated verification, by providing experimental results of checking these using a verification framework built on top of our Bogor software model checker (extended from our previous work on model checking JML [8] and atomicity specifications [9]).

Our approach does not explicitly deal with Java's relaxed memory model [10], and instead assumes a sequential consistent memory model. This assumption is sound for programs that are free of race conditions, and race-freedom can be verified via separate analyses [11].

Although we have used JML for this work, we believe the ideas could also be adapted to other specification languages such as Spec# and Eiffel. However, we leave detailed investigation of such adaptation for future work.

In the next section, we describe the problems addressed in this work, in particular, the limitations of JML for concurrent programs. Section 3 gives background on the concept of atomicity, on which our approach is based. Section 4 introduces the new set of annotations, giving examples and an assessment of the issues addressed by each annotation. Section 5 reports on an evaluation of using JML extensions to specify Java classes and of checking such specifications using a customized model checker. Section 6 surveys related work, and Sec. 7 concludes.

2 The Problem

In this section we discuss the semantical and expressiveness problems that need to be solved to allow effective specification and reasoning about both data values and proper thread behavior in a multi-threaded program.

2.1 Interference

Interference causes problems that affect modular reasoning about data values in multi-threaded programs. These semantical problems are best illustrated by an example.

Consider the method in Fig. 1. This is a method from a concurrent linked queue class, and is adapted from Lea's book [12]. This method extracts an element from the queue. The figure shows an invariant for the class at the beginning

```
public class LinkedQueue {
    protected /*@ spec_public non_null @*/ LinkedNode head;
    protected /*@ spec_public non_null @*/ LinkedNode last;
    //@ public invariant head.value == null;

    /*@ public normal_behavior
      @   requires head == last;
      @   assignable \nothing;
      @   ensures \result == null;
      @ also public normal_behavior
      @   requires head != last;
      @   assignable head, head.next.value;
      @   ensures head == \old(head.next) && \result == \old(head.next.value);
      @*/
    public synchronized Object extract() {
        synchronized (head) {
            Object x = null;
            LinkedNode first = head.next;
            if (first != null) {
                x = first.value;
                first.value = null;
                head = first;
            }
            return x;
        }
    }
```

Fig. 1. JML Specification for the method `extract()`

of the class declaration. Invariants must be satisfied by the instances of the class at every method's pre- and post-state. The figure also shows a behavioral specification of the method written in JML, without using any of the extensions proposed in this paper. JML's annotations are written as special Java comments that begin with an at-sign (`@`). The specification of the `extract()` method appears just before its header. This specification is comprised of two `normal_behavior` specification cases, each of which has a `requires` clause, which gives its precondition, an `assignable` clause, which gives a frame axiom, and an `ensures` clause, which gives its postcondition. The first case applies when the list is empty at the beginning of the method's execution (`head == last`), and the other when the list is non-empty (`head != last`). In the first case the method must return `null`, without assigning to any locations. Otherwise, the method updates `head` to the next node in the queue (`head.next`), and must return the object that was contained in that node (since `head` is a sentinal, as described by the invariant). To satisfy the invariant the method must also make the value of the new head `null`. The method may assign to both `head` and `head.next.value` to achieve this behavior.

The meaning of a JML method specification with two or more specification cases, combined with "`also`", is that the caller has to satisfy the disjunction of the preconditions in the given specification cases, and the implementation has to satisfy the postconditions of all specification cases for which the preconditions held [4, 13]. Thus, in Fig. 1 the caller has no obligations, since the disjunction of the preconditions of the two specification cases is the tautology "`head == last || head != last`".

Internal Interference. Although the specification given in Fig. 1 seems sensible, it is wrong for a multi-threaded environment, because it does not account for interference. An example of interference is depicted in Fig. 2. The queue is empty in the method's pre-state. Since this method is `synchronized`, the first instruction it executes is to acquire the lock on `this`. Then, in this trace, another thread is immediately scheduled that executes a call to method `insert(Object)`, which is synchronized on a lock other than `this` (to allow a fine grained degree of concurrency), and executes it to completion. When the `extract()` call in the first thread resumes the queue is no longer empty and the method will find a non-null element to extract. Since the list was empty in the pre-state, the first specification case should apply, but the interference causes a non-null value to be returned which violates the postcondition of that case.

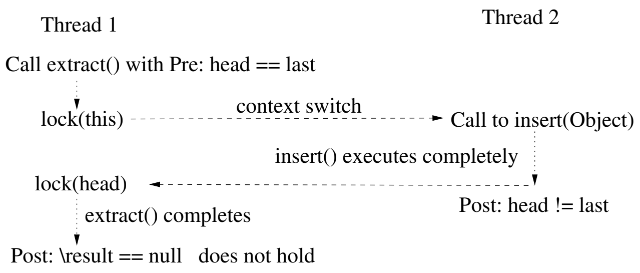

Fig. 2. Execution of `extract()` call interleaved with a call to `insert(Object)`

We call the problem illustrated by Fig. 2 *internal interference*. This problem arises when another thread affects the current thread's execution of a method, by changing data that the method can observe. Standard Hoare logic does not allow for such interference, and thus when reasoning about the correctness of the implementation of a method in Hoare logic, one assumes that properties, such as the method's precondition, do not change except by actions of the method itself.

Runtime and static analysis tools for sequential programs exploit this semantics when they work with just two states: the *pre-state* (at the beginning of the method's execution) and the *post-state* (at the end). In a multi-threaded setting, however, such analyses must consider all possible interleavings to safely account for possible interference. This is considerably more expensive than restricting reasoning to pre- and post-states; furthermore, it is non-modular.

External Interference. *External interference* happens when another thread makes observable state changes between a method call and the method's entry, or between the method's exit and the caller's resumption. Just as internal interference can invalidate standard Hoare-style reasoning about the correctness of a method implementation, external interference can disrupt reasoning about the correctness of client code that calls that method.

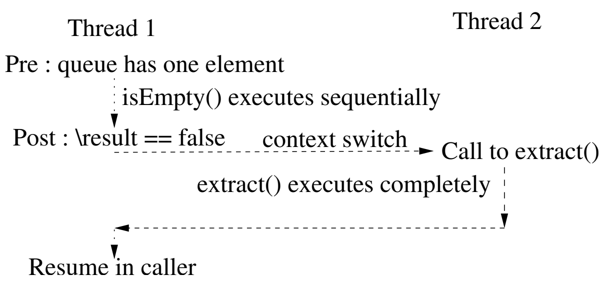
Thread 1
Thread 2
Pre : queue has one element
isEmpty() executes sequentially
Post : \result == false
context switch
Call to extract()
extract() executes completely
Resume in caller

Fig. 3. Execution of `extract()` call interleaved immediately after call to `isEmpty()`

Figure 3 illustrates the problem. Suppose the queue has exactly one element. The `isEmpty()` method executes without interleaving and returns false. However, between the return of `isEmpty()` and the resumption of the caller, another thread interleaves a call to `extract()`, which removes the lone element. The result is that upon resumption of the caller's thread, the postcondition no longer describes the queue correctly, as the queue is now empty. Similar interference can happen with respect to the precondition established by the caller and observed upon method entry. As can be seen from this example, external interference breaks the modularity of reasoning, thereby rendering existing sequential analysis techniques inapplicable.

More generally, these types of interference mean that specifications cannot serve as behavioral abstractions in the presence of concurrency. That is, one cannot use method specifications to reason about the correctness of an implementation without knowing some details of its calling context, or reason about calls to the method without knowing some details of its implementation. To fix these problems, while still allowing modular reasoning, the specification language must be enriched.

2.2 Expressing Thread-Safe Behavior

There are a variety of ways that one might incorporate features into a behavioral specification language to support specification in the presence of concurrency. Our approach has been motivated primarily by the results of a survey of existing Java implementations to understand the mechanisms used by developers to assure proper concurrent execution, and of existing specification features in the JML language. We wanted to minimally extend JML, and yet enable modular behavioral specification of a wide range of existing multi-threaded Java implementations.

Our fundamental observation is that while programmers may use a variety of mechanisms to achieve thread safety, the core notion of safety is one of non-interference. Interference can be avoided through the use of synchronization, which prevents unwanted interleaving, or by controlling access to data, which prevents unwanted access to otherwise shared objects.[1] When we speak of *thread safety*, we mean either synchronization or controlled access to data, or some combination of both. We have identified several fundamental notions that must be included in JML or similar languages to support thread-safe behavioral specification.

Locking Specifications. Programmers use a variety of locking disciplines and the language must be rich enough to capture that variety. It is necessary to allow specification of:

– what locks a method will acquire and release during its execution,
– what locks protect particular parts of an object's state,
– that some objects are used as locks, and when such lock objects are locked,
– the set of locks held by the current thread, and
– that an object is protected by some lock held by the current thread.

We have also found it necessary to specify the conditions under which a method may block [14].

Data Confinement Specifications. Excessive locking can reduce parallelism and hence performance. For this reason, many implementations avoid locking and instead rely on properties of a program's data layout to ensure thread safety. It is necessary to allow specification of:

– what aliasing and ownership patterns exist among objects [15, 16, 17],
– that an object is local to a thread, and
– the effect of a method's execution on existing locations (*i.e.*, a frame axiom [18]).

Serializability Specifications. Locking and data confinement specifications are useful in specifying the conditions under which multi-threaded executions are equivalent to sequential executions. It is necessary to allow specification of this high-level *serializability* property of methods. We have found two different strengths of such specification to be useful in practice: atomicity and independence. These concepts and the extensions to JML that support them are described in the next sections.

3 Background on Atomicity and Independence

Our approach to addressing the interference problems above is based on the concepts of atomicity [19, 20] and independence [21]. A region of code statements (*e.g.*, a method body) is said to be *atomic* if the statements in the region are *serializable*—that is, if for any execution trace containing the region's statements

[1] We do not treat extra-program forms of concurrency control, such as scheduling.

(possibly interleaved with statements executed by other threads) there is an equivalent execution trace where the region's statements are executed sequentially (*i.e.*, executed without any interleavings from other threads). If a code region is atomic, then it is sound to reason about its actions as if they occur in a single atomic step—in essence, allowing one to use traditional sequential reasoning techniques on the code region. From another point of view, instead of having to consider a number of intermediate states produced by thread interleavings, for an atomic region it is sound to consider only two states: the *pre-state* before the conceptual single atomic step begins, and the *post-state* after the conceptual single atomic step completes. That is, any interference from the other threads is benign, however, the single atomic step may interfere with other threads' computations.

There are many ways to establish the atomicity of a code region. In the next two subsections we describe two popular approaches; these approaches will motivate the JML notations that we develop in the following sections.

3.1 Lipton's Reduction Theory

Lipton introduced the theory of left/right movers to aid in proving properties about concurrent programs [19]. In Lipton's model, a code region is thought of as a sequence of primitive statements (*e.g.*, Java bytecodes), which he called *transitions*. Proofs about the transitions in a program can be made simpler if one is allowed to assume that a particular sequence of transitions is indivisible. To conclude that a program, P, which contains a sequence of transitions, S, is equivalent to the reduced program, P/S, in which S is modeled as one indivisible transition, Lipton proposed the notion of a commuting transition. A *commuting transition* is a transition that is either a right mover or a left mover. Intuitively, a transition, α, is a *right (left) mover* if, whenever α is followed (preceded) by another transition, β, of a different thread, then α and β can be swapped without changing the resulting state. Concretely, a lock acquire, such as the beginning of a Java synchronized block, is a right mover, and the lock release at the end of such a block is a left mover. Any read or write to a variable or field that is properly protected by a lock is both a left and right mover, which is termed a *both mover*.

To illustrate the application of these ideas, we repeat the example given in [20]. Consider a method m that acquires a lock, reads a variable `x` protected by that lock, updates `x`, and then releases the lock. Suppose that the transitions of this method are interleaved with transitions E_1, E_2, and E_3 of other threads, as shown at the top of Fig. 4. Because the actions of the method m are movers (`acq` and `rel` are right and left movers, respectively, and the lock-protected assignment to `x` is a both mover), Fig. 4 implies that there exists an equivalent execution (shown at the bottom of the figure), where the operations of m are not interleaved with operations of other threads. Thus, it is safe to reason about the method as executing in a single atomic step.

One can define an *atomic region* as one that satisfies the pattern of statements $\mathsf{R}^*\mathsf{N}^?\mathsf{L}^*$, where R^* denotes 0 or more right mover statements, L^* denotes 0 or more

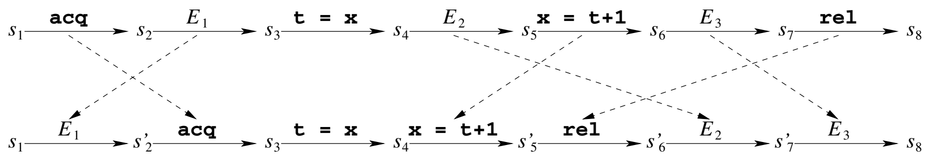

Fig. 4. Left/Right movers and atomic blocks

left mover statements, and $\mathsf{N}^?$ denotes 0 or 1 statements that are neither left nor right movers. That is, an atomic region can contain at most one non-commuting (*i.e.*, possibly interfering) statement. The block shown in Fig. 4 matches this pattern in the following way: the statement `acq` is right mover (R), the statement `t = x` is both right and left mover so it can stand for right (R), the statement `x = t+1` is both right and left mover so it can stand for left (L), and the statement `rel` is left mover. Thus we get RRLL, which fits the pattern.

In other words, an atomic region can have a single externally-observable effect in its body while it is executed. However, note that an atomic method can have multiple accesses to heap objects as long as they are either thread local or lock-protected accesses [22]. This is because accesses to objects local to a thread and to objects protected by locks cannot be observed by other threads until these objects become shared or until the locks are released, respectively. Similarly, lock-acquires and lock-releases on an object that is already locked by a thread cannot be observed by other threads until that lock is released.

Lipton also stated two technical conditions necessary to prove that the set of final states of a program P equals the set of final states of the reduced program P/S. The first of these, R1, [19–p. 719] states that "if S is ever entered then it should be possible to eventually exit S." This is a fairly strong liveness requirement that can be violated if, for example, S contributes to a deadlock or livelock, or fails to complete because it performs a Java `wait` and is never notified. Restriction R2 is that "the effect of statements in S when together and separated must be the same." This is essentially stating an interference-free property from the other threads for S: any interleavings between the statements of S do not affect the final store it produces.

3.2 Independent Statements

Statements that are both movers are also referred to as *independent statements* since they can commute both to the left and the right of other program statements. We say that a code region is *independent* if each statement within the region is independent. Thus, an independent code region satisfies the pattern of statements I^*, where I^* denotes 0 or more both mover statements. An independent region is totally non-interfering, and thus is trivially atomic.

Our specification methodology for reasoning about method calls within contexts will rely on the fact that independent regions have pleasing composability properties. Sequentially composing two independent regions yields an independent region, but composing two atomic regions may not yield an atomic region if each region contains a non-mover. Moreover, the sequential composition of an independent region and an atomic region is an atomic region (since both movers can serve as either left or right movers). Intuitively, calling a method M_2 from inside of a method M_1 represents the sequential composition of three code regions (the part of M_1 before the call, the body of M_2, the rest of M_1). Thus, if one takes an atomic method, M_1, and inserts into its body a call to an independent method, M_2, then M_1 remains atomic. However, if the inserted call is to an atomic method, M_3, then M_1 does not necessarily remain atomic.

Finally, we note that for a region to be independent, one does not have to establish Lipton's R1 liveness condition to ensure the existence of an equivalent serialized trace (it is the asymmetric nature of the left and right movers in the criterion for atomic regions that necessitates the liveness condition, R1).

4 Introducing Concurrency into JML

Our approach for introducing concurrency into JML is to separate the concern of property specification for methods into two parts: (1) atomicity and independence properties, and (2) specification of sequential (or functional) behavior. This is an old idea, but quite useful. What we claim is new is the language design, which provides necessary and sufficient constructs to specify a wide range of multi-threaded programs.

In what follows we describe just the new JML constructs relating to specification of atomicity and independence and to expressing locking and other properties specific to multi-threaded programs. We illustrate the new constructs using examples from Doug Lea's and Java 1.5's concurrent libraries.

4.1 Locking Notations

Locking is an important aspect of concurrent behaviors, since it is the usual mechanism used to achieve atomicity. So we need several notations that allow for the specification of locking behaviors. These notations also allow modular checking of atomicity specifications (described in Sec. 4.3).

JML already has several notations that can be used to specify information about locking. The `monitors_for` clause allows specifying the locks that protect the access to a given field. The syntax of this clause is:

⟨monitors-for-clause⟩ ::= `monitors_for` ⟨ident⟩ `<-` ⟨store-ref-list⟩ `;`

The meaning is that all of the (non-null) locks named in the ⟨store-ref-list⟩ must be held by a thread in order to access the field `ident`. (In JML, a ⟨store-ref-list⟩ is a comma-separated list of access expressions, which includes identifiers, field and array accesses, and various patterns [4].) An example in Sec. 4.4 demonstrates the use of the `monitors_for` clause.

Finally, the `\lockset()` expression returns an object, of type `JMLObjectSet`, that represents the set of all locks held by the current thread.

The first new construct we add to JML is the `locks` clause. This clause can appear in the body of a specification case after the heading part (in which the `requires` clause appears). Its syntax specifies a list of locks:

⟨locks-clause⟩ ::= `locks` ⟨store-ref-list⟩`;`

Figure 5 shows the use of the `locks` clause in the method `extract()`. The `locks` clause accomplishes two different purposes. First, it is an explicit statement of the locks that the current method acquires (and releases) during its execution. The meaning is that, on any given execution (where the precondition is satisfied), the method will lock all the locks in the given list. Second, the `locks` clause states an implicit condition for independence. In general, a `locks` clause of the form:

`locks` $l_1, \ldots, l_n$`;`

desugars to an ensures clause of the form:

`ensures \old(\lockset().has(`l_1`) && ... && \lockset().has(`l_n`))`
`        ==> \independent;`

Therefore, if another method calls the method with the `locks` clause in a context where all the locks in the list are held, then the callee must be guaranteed to be independent (see Sec. 4.4). In this sense, the `locks` clause gives a lower bound on the set of locks that must be held by callers to ensure independent execution when the method is called. This can help verify a caller's atomicity specification. Conversely, the `locks` clause limits the locks that an implementation of the method may try to acquire (when the precondition of the specification case it appears in holds); thus for the implementation the `locks` clause gives an upper bound on the set of locks that the method may try to acquire.

For an instance (static) method, the `locks` clause has a default value of `this` (the class object) if the method is specified as `synchronized`, otherwise it defaults to `\nothing`. The default for synchronized methods is useful because many concurrent methods in Java synchronize on `this`.

Finally, we add a predicate, `\lock_protected`, with the following syntax.

⟨lock-protected-expression⟩ ::= `\lock_protected(`⟨store-ref⟩`)`

An expression such as `\lock_protected(`o`)` states that the object referenced by o is access-protected by some nonempty set of locks, and all of those locks are held by the current thread. Notice that this is a very strong property: the access is restricted with respect to the object, not the reference variable, therefore if the object is aliased, this property states that access to all the aliases is restricted by the lock-set of this object. The identities of these locks are not specified. This notation allows one to specify locking behavior, while hiding the details of locks involved. Verification of `\lock_protected(x.f)` would use the `\monitors_for` clause for `f` in `x`'s class.

4.2 Heap Restriction Notations

In addition to locking, thread safety can also be achieved by restrictions on references. JML's heap restriction notations are aimed at specifying how local

```
public class BetterLinkedQueue {
    protected /*@ spec_public non_null rep @*/ LinkedNode head;
    protected /*@ spec_public non_null rep @*/ LinkedNode last;
    //@ public invariant head.value == null;

    /*@ public normal_behavior
      @   requires head == last;
      @   locks this, head;
      @   assignable \nothing;
      @   ensures \result == null;
      @ also public normal_behavior
      @   requires head != last;
      @   locks this, head;
      @   assignable head, head.next.value;
      @   ensures head == \old(head.next) && \result == \old(head.next.value);
      @*/
    public /*@ atomic @*/ synchronized /*@ readonly @*/ Object extract() {
        synchronized (head) {
            /*@ readonly @*/ Object x = null;
            /*@ rep @*/ LinkedNode first = head.next;
            if (first != null) {
                x = first.value;
                first.value = null;
                head = first;
            }
            return x;
        }
    }
```

Fig. 5. Extended JML specification for `extract()`

variables may refer to objects in the heap, and how these objects may refer to each other. They allow dealing with issues like representation exposure [16] and other kinds of unwanted aliasing that would otherwise prevent modularly checking atomicity specifications. For example, consider the method `extract()` in Fig. 1. This method accesses the field `head` by first acquiring the lock on the object, so as to ensure atomicity. However, if there is representation exposure, in particular if there is another reference to the object pointed to by `head`, then that alias might be held by another thread. Thus one would have to examine other code in the program to rule out access by some other thread to the state of the object `head` refers to, in particular to the field `head.next`. In other words, representation exposure of this sort would necessitate a non-local analysis of the program to rule out such possible interference.

To prevent these problems we take advantage of the Universe type system [17], an ownership type system that already exists in an experimental form in JML [23]. This type system adds the modifiers `rep` and `readonly` to declarations.

The `rep` modifier can be used on field declarations. It states that the object referenced by the specified field is part of the representation of the given class. There can be no references from outside an object of the class to such representation objects. From outside the class, one can only refer to the enclosing object, which is the *owner* of the representation objects. For example, in Fig. 5 the fields `head` and `last` are `rep` fields, therefore there can be no external aliases to the objects to which these fields refer, and hence no representation exposure. This enables the modular verification of the atomicity specification.

```
/*@ normal_behavior
  @   requires c != null && \thread_local(c);
  @   assignable elementCount, elementData;
  @   ensures elementCount == c.size() && \fresh(elementData);
  @also
  @ exceptional_behavior
  @   requires c == null;
  @   assignable \nothing;
  @   signals (Exception e) e instanceof NullPointerException;
  @*/
public /*@ atomic @*/ Vector(Collection c) {
  elementCount = c.size();
  elementData = new Object[(int)Math.min((elementCount*110L)/100,Integer.MAX_VALUE)];
  c.toArray(elementData);
}
```

Fig. 6. Extended JML specification for a constructor in `java.util.Vector`

The `readonly` modifier is a type modifier. It marks a reference as read-only, meaning that the object cannot be modified through that reference. Read-only references are not necessarily owned by an object containing the `readonly` field, and it is often the case that such references are aliased externally. The idea is that only the identity of a `readonly` object matters to the abstract state of the enclosing object. (JML will eventually enforce various restrictions on access to read-only objects in assertions.)

The notations just discussed deal with ownership between objects. Equally important in concurrent programs is the ownership of object by threads. An *object o is owned by a thread t* if only thread t can reach o by a reference chain. This condition guarantees that there cannot be a race condition on o, because o is not shared. We introduce the notation `\thread_local(`o`)` with the meaning that o is owned by the current thread. The general syntax is as follows.

$\langle$thread-local-expression$\rangle$ ::= `\thread_local ( `$\langle$store-ref$\rangle$` )`

This notation is useful for modular verification of atomicity, because accesses to thread local objects are independent (non-interfering).

For example, consider the constructor from Java's `Vector` class shown in Fig. 6. In general, constructors are independent because the constructed object is not reachable from any other thread. However, if the constructor takes object arguments to initialize the internal state of the constructed object, then its execution might not be atomic. The problem is that such an argument object might be concurrently modified by other threads. So, in this example the constructor's precondition requires that that the argument, `c`, be thread local.

4.3 Atomicity Modifier

We introduce atomicity specification into JML with a new method modifier, `atomic`. This specifies that, when a method is invoked in a state that meets its precondition, its implementation must ensure that the resulting execution is serializable. This modifier is inherited by overriding methods. Fig. 5 shows how we use this new modifier to specify `extract()` from Fig. 1.

```
public class ArrayBlockingQueue<E> {
  private /*@ spec_public non_null rep @*/ final E[] items;
  //@ monitors_for items <- lock;
  private /*@ spec_public rep @*/ final ReentrantLock lock;

  /*@ normal_behavior
    @   requires lock.isLocked() && 0 < i && i < items.length;
    @   ensures \result == \old((i + 1) % items.length) && \independent;
    @*/
  final /*@ atomic @*/ int inc(int i) {
    return (++i == items.length)? 0 : i;
  }
```

Fig. 7. Extended JML specification for `inc()`

Checking that a method declared to be atomic is actually atomic can be done in a variety of ways. For example, one could prove that the code is reducible by Lipton's theory [20, 24] or by using the notion of independent transitions. Another technique is used in the Atomizer [25], which dynamically checks that lock acquisitions and releases are properly nested and that all accesses to shared data is lock protected. The `monitors_for` clause would be used to determine what locks protect what pieces of data.

By imposing an additional obligation to guarantee serializable executions on a method's implementation, the `atomic` modifier simplifies the implementation's proof of functional correctness. The functional correctness proof can assume that the execution is serializable, thus avoiding internal interference. For example, in the method `extract()` specified in Fig. 5, the postcondition of each specification case must hold only for traces in which the method is executed sequentially. However, when combined with the additional proof of atomicity, one still gets a strong correctness guarantee about the complete behavior of the implementation. This division of proof obligations for implementations allows proofs of functional correctness to be separated from synchronization details.

From the caller's point of view, there is no potential for internal interference by atomic methods, and thus the caller only has to worry about external interference. To avoid external interference, the caller must ensure that objects needed to preserve the truth of any precondition or postcondition are thread safe (*e.g.*, locked or local to the caller's thread [22]). This additional requirement helps shake out synchronization bugs without heavyweight temporal logic—this is something that has not been explored in other static/dynamic analyses for atomicity.

It is also possible for an atomic method to transfer some of its obligation to ensure atomic execution to the caller, by stating a precondition involving locks or thread ownership. For example, an atomic method can require some locks to be held before being called, using a precondition such as `\lockset.has(lock)`, which says that the current thread holds the lock named `lock`. Such a precondition may have the added benefit of preventing external interference. Indeed, having the ultimate clients obtain locks may be sensible from an overall design standpoint (via an end-to-end argument [26]). Figure 7 show an example of this case, in which the method `inc()` transfers the responsibility of holding the lock on `lock` to the caller. (This ability to transfer some obligation to the caller shows how `atomic` is different than Java's `synchronized` modifier.)

Finally, in many concurrent classes, all methods should be atomic. To allow a designer to state this, the `atomic` keyword can be used in a class or interface declaration. Such a modifier simply states that all the methods declared in the type's declaration are atomic. This type modifier is inherited by subtypes.

4.4 Independent Predicate

A method execution is *independent* if all of its transitions are independent. For example, accesses to objects local to the thread and accesses to objects that are protected by locks are all independent transitions, since the other threads cannot observe such accesses. To specify this property of a method execution, we introduce a new specification predicate, `\independent`. This predicate can only be used in postconditions.

An example that shows how independence can be used to avoid external interference is `java.util.concurrent.ArrayBlockingQueue`'s method `inc()`. This method is specified in Fig. 7. The precondition states that the method must be called from a context in which `lock` is locked. Since `items` is protected by `lock`, due to the `monitors_for` declaration, and since `i` is a parameter of the method, no other thread can access the data used by `inc()` and cause any interference. Thus, when the precondition is met, this method's executions are independent. Furthermore, since the caller must hold the lock, and since the result is not accessible to other threads, calls cannot suffer external interference.

4.5 Blocking Behavior and Commit Atomicity

In this section we describe notations for handling methods that wait for some condition to become true before they proceed to take some (atomic) action. Such methods have what we call a *blocking behavior.*

Consider the method `take()`, from `ArrayBlockingQueue` in Java 1.5, as shown in Fig. 8. This method takes an element from the queue, but if the list is empty, it waits until there is an element to remove.

To specify the blocking behavior of methods, we use JML's `when` clause (adapted from Lerner's work [14]):

$\langle$when-clause$\rangle$ ::= `when` $\langle$predicate$\rangle$;

Its meaning is that, if a method is called in a state in which the method's `when` predicate does not hold, the method blocks until this predicate is satisfied (presumably by an action of a concurrent thread). This specification does not constrain what protocol is used to wait for the condition to become true; for example, a busy wait loop might be used. A blocking method specification can be formalized as a partial action that does not execute until the `when` predicate holds, and then atomically transitions from the pre-state to the post-state (as specified by the pre- and postcondition, *etc*). The `when` clause by default has a value of `true` (for JML's heavyweight specification cases).

```
/*@ public normal_behavior
  @   locks this.lock;
  @   when count != 0;
  @   assignable items[takeIndex], takeIndex, count;
  @   ensures \result == \old(items[takeIndex]) && takeIndex == \old(takeIndex + 1)
  @        && count == \old(count - 1);
  @*/
public /*@ atomic @*/ E take() throws InterruptedException {
  final ReentrantLock lock = this.lock;
  lock.lockInterruptibly();
  try {
    try {
      while (count == 0)
        notEmpty.await();
    } catch (InterruptedException ie) {
      notEmpty.signal(); // propagate to non-interrupted thread
      throw ie;
    }
    /*@ commit: @*/ E x = extract();
    return x;
  } finally {
    lock.unlock();
  }
}
```

Fig. 8. Extended JML specification for `take()`

To check `when` clauses, we use a special statement label, `commit`. If this label is not present in a method, it is implicitly assumed at the method body's end. This label gives a *commit point* for the method; when execution reaches the commit point, the method is no longer blocked and the rest of the method's execution must be atomic. Also, the predicate given in the `when` clause must hold at the commit point, but not necessarily during the rest of the method's execution. This idea is related to the concepts of commitment in database transactions and the notion of "commit atomicity" introduced by Flanagan [27].

Figure 8 illustrates the use of the `when` clause and the `commit` label. Method `take()` removes an element from the queue, and blocks if the queue is empty. The `when` clause in Fig. 8 says that the method may proceed only when `count` is not zero. The commit point of this method is where the `commit` label appears, right after the loop that blocks until the queue is non-empty.

4.6 Notations for Lock Types

Another important set of notations has to do with identifying what classes and interfaces have instances that are intended to be locks. Such lock objects are an addition to the implicit reentrant lock in each object that can be acquired using Java's `synchronized` statement. Java 1.5 adds several such types of lock objects, which support new concurrency patterns. An example is the new class `ReentrantLock`, whose instances are specialized locks that are manipulated by method calls. Such locks can make synchronization more flexible and allow for more efficient code. In JDK 1.5, users can also define their own locks by implementing the interface `java.util.concurrent.locks.Lock`.

To deal with these new kinds of locks, JML will consider a type to be *lock type* if it is a subtype of the `Lock` interface. An expression whose static type is a lock type is considered to denote a *lock object*. There is a potential semantic ambiguity that arises because lock objects also contain Java's implicit synchronization locks. To resolve this ambiguity we assume that when lock objects are mentioned in a context where a lock is expected, the specifier always means the lock object itself, not the implicit synchronization lock it contains. For example, in a `monitors_for` clause a lock object expression refers to the lock object itself.

To know when a lock object is locked, we introduce a new type-level declaration, the `locked_if` clause. Its syntax is as follows:

⟨locked-if-clause⟩ ::= `locked_if` ⟨predicate⟩ ;

Each type that is a subtype of `java.util.concurrent.locks.Lock` must declare or inherit exactly one `locked_if` clause. This clause states a predicate that holds if and only if the given instance of the lock type is in the *locked* state.

So, for example, for the class `ReentrantLock` we have:

```
package java.util.concurrent;
import java.util.concurrent.locks.Lock;
public class ReentrantLock implements Lock, java.io.Serializable {
   //@ locked_if isLocked();
   /* ... */
}
```

In `ReentrantLock`, `isLocked` is a (pure) method that returns `true` if the target instance is locked, and `false` otherwise. The `locked_if` clause is used in the `lockset()` operator's semantics. For example, if `rl` has type `ReentrantLock`, then `\lockset().has(rl)` returns `true` if and only if `rl.isLocked()` holds.

4.7 Revisiting External Interference

In Sec. 4.3 we described how the `atomic` modifier solves the internal interference problem by providing an abstraction in which other threads did not interleave during the execution of the method. Now we look at how the rest of the annotations help in solving the problem of external interference. As suggested in Sec. 4.3, the key to preventing this problem is disallowing access from other threads to the objects mentioned in the pre- and postconditions.

External interference is a problem of interference between two methods: one method (the caller) calling another method (the callee) and other threads breaking the contract between the two of them. To support contract specifications that account for external interference we consider two cases: an atomic method calling another atomic method, and a non-atomic method calling an atomic method. Note that atomic methods can only call atomic methods, and the case where a non-atomic method calls another non-atomic method, aside from being uncommon, would not be handled by our notations.

In the first case, if an atomic method calls another atomic method, then the interference between the caller and the callee would be internal interference in the caller, which is already handled by the atomicity abstraction.

In the second case, when a non-atomic method calls an atomic method, the caller needs to ensure that the objects needed to preserve the truth of the pre-

and postcondition are *thread safe.* We define thread safety by introducing a new operator, `\thread_safe`, defined such that

$$\texttt{\textbackslash thread\_safe}(SR) \equiv \texttt{\textbackslash thread\_local}(SR)\ \texttt{||}\ \texttt{\textbackslash lock\_protected}(SR).$$

That is, SR is thread-safe if it is owned by the current thread or is lock protected. To avoid external interference, a contract must require that all objects needed to preserve the truth of the pre- and postcondition are thread-safe. While this is a strong condition, in our experience it is satisfied by all well-written multi-threaded code.

As an example of thread safety let us take another look at Fig. 6. In that example, the specification requires the collection `c` to be thread local, however this condition is actually stronger than what is actually needed. The actual requirement is that the collection be free of interference. Therefore, we can relax the precondition in the `Vector` constructor to `\thread_safe(c)`, accounting for the instances in which the constructor is called with an argument that is externally protected by a lock.

5 Evaluation

In this section we describe our experiences in applying our JML extensions. We evaluate their adequacy and efficacy by applying them in a collection of *specification case studies.* In these studies, we attempt to write complete behavioral specifications for a collection of Java classes drawn from the literature that were designed explicitly for use in multi-threaded programs. These classes use significantly more complex concurrency policies than do typical classes, *e.g.*, Java container classes. Thus, if we can support the specification of rich functional properties for these classes, then our extensions will be broadly applicable. We also evaluate the checkability of our JML extensions in a set of *verification case studies.* In these studies, we sampled the classes and specifications from our specification studies and checked them using an extension of the Bogor model checking framework [28] described below.

The next two sections present the details of these of case studies, give a summary of the results obtained, and an account of our conclusions. The complete set of artifacts used in our studies are available from the web [7].

5.1 Specification Case Studies

To assess the adequacy and behavior coverage of the extensions to JML, we identified a set of of concurrent Java classes and wrote specifications for their methods using the extended JML. The classes come from multiple sources and most are implementations of concurrent data structures:

– A bounded buffer, `BoundedBuffer` (from Hartley [29]).
– Dining philosophers, `DiningPhilosophers` (from Hartley [29]).
– A linked queue, `LinkedQueue` (from Lea [12]).

Table 1. Summary of statistics from specification case studies with the extended JML. The classes marked with a * belong to Java 1.5's package `java.util.concurrent`

Class Name	Number of methods	Frequency of annotations				
		`atomic`	`\independent`	`locks`	`\thread_safe`	`when`
`BoundedBuffer`	3	3	0	2	0	2
`DiningPhilosphers`	7	7	4	2	0	1
`LinkedQueue`	7	7	0	7	0	1
`RWVSN`	8	8	2	4	0	2
`java.util.Vector`	45	45	4	34	9	0
`ArrayBlockingQueue`*	19	19	7	15	3	2
`CopyOnWriteArrayList`*	27	27	6	13	12	0
`CopyOnWriteArraySet`*	13	13	2	6	5	0
`DelayQueue`*	17	17	3	14	4	2
`LinkedBlockingQueue`*	17	17	4	12	1	2
`PriorityBlockingQueue`*	21	21	4	10	1	1
`ConcurrentLinkedQueue`*	11	11	2	0	2	4
Total:	**195**	**195**	**38**	**119**	**37**	**17**

- Code for readers-writers, `RWVSN` (from Lea [12]).
- The class `java.util.Vector`.
- Eight concurrent classes from `java.util.concurrent` in Java 1.5.

The 8 classes from `java.util.concurrent` are particularly important, as they have fairly complex and varied concurrency patterns and represent the new Java concurrency paradigm.

Table 1 presents statistics on the specifications we developed. The data shown is only for the 195 public methods in the studied classes; including private methods brings the total to over 220. We note that for all methods we were able to write complete behavioral specifications. So, for this challenging set of concurrent classes, our extensions appear sufficient for capturing their behavior.

Table 1 also reports the frequency with which we used different groups of extended JML primitives; these groups were described in the sub-sections of Sec. 4. Each entry shows the number of methods in the class whose specification used an annotation in the given group.

We observe that all of the methods studied had specifications that used the keyword `atomic`, that is, the methods exhibit the atomicity property. These results add to existing evidence [25] in support of the conclusion that most Java methods are intended to execute atomically. This validates our approach to using atomicity as the central abstraction for extending JML to support concurrency.

The use of `\independent` is not particularly common in this collection of classes. We believe that this is due to the fact that methods in these classes generally have complex concurrency policies. More typical classes with *get* and *set* methods for instance fields would probably yield large numbers of independent methods, but a broader study of Java classes is needed to confirm this intuition.

The study confirms the popularity of synchronization in enforcing correct thread-safe class behavior as more than 60% of the methods used the locking extensions. Use of data confinement is much less common in this study with less than 20% of the methods using `\thread_safe` annotations.

Table 2. Summary of statistics from verification case studies with the extended JML. Classes marked with a * belong to Java 1.5's package `java.util.concurrent`

Class Name	Number of Methods	Checkable Atomicity	Checkable Functionality	Coverage Ratio
`BoundedBuffer`	3	1	3	.67
`DiningPhilosphers`	7	6	7	.93
`LinkedQueue`	7	1	7	.57
`RWVSN`	8	4	8	.75
`CopyOnWriteArrayList`*	10	5	10	.75
`LinkedBlockingQueue`*	7	3	7	.71
Total:	**42**	**20**	**42**	**.74**

We believe that the sparse use of `when` clause in this study is due to the fact that most of our classes are container data structures. In most cases, concurrent data structures have two blocking methods: one that inserts elements but blocks if the structure is full and another one that removes elements but blocks if the structure is empty. More varied interfaces for accessing and modifying stored data will increase the need for this annotation.

The most important result of these studies is the fact that the proposed JML extensions appear to be both necessary (all annotations are used in the study) and sufficient (all methods in the study could be specified) for supporting thread-safe functional specification.

5.2 Verification Case Studies

In previous work [28], we showed how an extensible model checking framework, called Bogor, could be extended to check complex JML specifications [8], and how that framework could be independently extended to check atomicity specifications [9]. We have integrated these two separate extensions to execute simultaneously during state-space analysis to check extended JML concurrency specifications. The main technical novelty of this integration is that postcondition and frame condition checking is enforced only if the current execution of the method was serial, that is, if the method body was executed without any interleaving from other threads. So this strategy both checks the functional specifications and independently assures that all concurrent runs of the method conform to the atomicity specifications. If either the atomicity specification or the functional specification are not satisfied, then Bogor reports a specification violation.

The result of applying this specification checking tool to a subset of the classes listed in Table 1 are summarized in Table 2. The first column in this table displays the class name for the particular case study. The second column shows the total number of methods involved in the case study. For some classes, only a fraction of the total methods in the class were checked. We selected methods with diverse functionality instead of checking large numbers of similar methods.

The rest of the columns in the table present data on the degree to which the tool was capable of reasoning about specified methods. We divided the specification into two parts: the atomicity specification and the functional specification. The third column in the table shows the number of methods in the class for which the atomicity specification could be checked, and the next column shows the number of those for which the functional specification could be checked. Finally, the last column gives a ratio of specifications checked versus total specifications written for all methods in a class.

Table 2 shows that the tool could verify all of the functional specifications for each method in the study. We note that these are strong specifications that involve quantification over heap elements, checking frame conditions, freshness, reachability, and calculating the values of memory locations in the pre-state.

Checking of atomicity is not nearly as complete. The tool could verify atomicity for only 20 out of the 42 methods. The study included 22 methods that exhibit a kind of atomicity which Bogor cannot verify. Bogor's atomicity checking mechanism is based on Lipton's reduction [19] and transition independence [9], whereas the 22 uncovered methods in these case studies exhibit a different type of atomicity. 11 of those 22 methods exhibit *commit atomicity* as defined by Flanagan in [27]. In that work, Flanagan described a model checking algorithm that allows checking commit atomicity specifications. This technique could be integrated into Bogor, and would yield a coverage ratio of .87.

The other 11 uncheckable methods implement complex concurrency patterns that our tool could not detect, even if enhanced to detect commit atomicity. The model checker could be further extended, of course, to include these synchronization patterns and thereby increase checking coverage. But since checking atomicity is undecidable in general, there will always be some patterns that the tool could not detect. Fortunately, the complex patterns and challenging concurrency classes that we selected for our study are not common in real application code. Indeed, Flanagan and Freund found that more than 90% of the methods they analyzed [25] exhibited relatively simple forms of atomicity. Thus we expect that many real programs specified with extended JML will be amenable to analysis via model checking. However, a significantly broader evaluation of the use of JML and its support for analysis will be needed to confirm this conjecture.

6 Related Work

Perhaps the closest related work to ours is the work on extending Spec# to deal with multi-threaded programs [30]. The specification language part of Spec# [5], is similar in many ways to JML, although it is integrated into the programming language (as in Eiffel [6]). Like JML, Spec# also has an extensive tool set, including runtime assertion checking and a verification engine. Although Spec# is very similar to C#, it is a new programming language that extends and modifies C# in several ways. The most interesting of these changes to C# come in the ways that Spec# deals with alias control and concurrency control. In both of these areas, Spec# uses new statements (`pack` and `unpack` for alias control, and `acquire` and `release` statements for concurrency control). The treatment of alias control is more dynamic than that found in the Universe type system which JML uses, which may make it more difficult to analyze statically. For concurrency control, Spec# deals with external interference in a drastic fashion, by having `acquire` gain exclusive access to an object, so that it is thread local. The Spec# discipline solves the internal and external interference problems, and has a proof of soundness. However, the approach only applies to programs that can be written following that discipline. The authors list as future work "extending the approach to deal with other design patterns" [30–Sec. 9]. In contrast, our work attempts to deal with existing concurrent Java programs, without requiring that they follow a particular programming discipline.

Ábrahám *et al.* [31] provide a proof system for multi-threaded Java programs. Their analysis is sound and tool supported. However, as they rely on whole-program Owicki-Gries style annotations they do not achieve modularity in the sense we aim for (*i.e.*, at the level of individual compilation units). Furthermore, their proof system only deals with monitor synchronization, whereas our approach is applicable to all Java, and accepts a very wide range of synchronization patterns by abstracting away from synchronization conditions. Thus our approach promises to be more useful for existing Java code.

Robby et al. [8] identified the problem of *internal interference* described in Sec. 2. They solved it by refactoring the functional code of a method, into another method, separating it from the synchronization code. In this way they are able to check JML specifications upon the refactored method that is always called within an atomic context. However, this technique is both limited in its applicability and inconvenient for users.

Freund and Qadeer implement a modular analysis for atomic specifications on multi-threaded software [32]. The idea of using a label to mark the commit points of a method, similar to the `commit` label introduced in Sec. 4.5, comes from their work. They achieve modularity by annotating shared variables with an access predicate, and by using the concepts of reduction to link a procedure to its specification. They translate a multi-threaded program into a sequential program in which atomic procedures are executed sequentially. However, JML is more expressive than the specification language they used.

Hatcliff et al. [9] developed a technique to verify atomicity annotations using model checking. Wang and Stoller [33] provide two atomicity detection algorithms based on runtime analysis: one based on Lipton's reduction [19] and another based on a sophisticated pattern matching mechanism. However, this system only provides verification of atomicity specifications. The verification tool described in Sec. 5 can be viewed as a natural extension to these techniques, which also checks functional specifications.

7 Conclusions and Future Work

We have extended JML by adding notations that allow the verification of multi-threaded Java programs. The overall approach is to use the concept of atomicity, from Lipton's reduction theory for parallel programs. We have shown how the added annotations support the concept of atomicity and allow the specification of locking behavior. In addition, we have shown how the concept of atomicity can be used to avoid the problems of internal and external interference, and thus to support modular reasoning. We have described our success in writing extended JML specifications of existing Java classes and have reported results on the implementation of a tool that leverages these language extensions to verify behavioral specifications of multi-threaded programs.

We are planning on extending and continuing this work along several lines. On the JML language side, we are planning to work on a formalization of all the new language constructs presented in this paper, and introduce a formal modular analysis for behavioral specifications of multi-threaded programs. Some details, such as how to extend JML's concept of pure methods to allow for locking [3] also need to be worked out. Also, we are studying other ways to improve concurrency support in JML. For example, one way in which JML could be further improved is the addition of temporal logic specification operators based on specification patterns [34] as in BSL (Bandera Specification Language) [35]. There are synchronization pattern implementations, such as those presented in [36], for which it is not clear whether the extensions presented in this work are sufficient, and that might require JML to be extended with temporal logic annotations to be properly specified.

On the tool support side, the same basic division of labor described in Sec. 5 for model checking, could be used to adapt JML's runtime assertion checking tool [37] to our JML extensions. This tool instruments Java programs with additional instructions that check method pre- and postconditions, invariants, *etc.* The idea would be to add checks from the Atomizer tool [25], which checks that program traces conform to Lipton's atomicity pattern. By separately checking for atomic executions, the runtime assertion checker could carry on as before, assuming that atomic methods were executed sequentially.

We plan to integrate this work into the JMLEclipse framework [38] which is an Eclipse-based front-end for JML verification engines (in particular, it will be the front-end for our JML model checking tool). Another possible path for future work is to extend other JML tools, such as ESC/Java2 or other verification tools (*e.g.*, [39]) to incorporate the new features.

Acknowledgments

This work was supported in part by: the U.S. Army Research Office (under grant DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by NSF (CCF-0306607, CCF-0341179, CCF-0428078, CCF-0429149, CCF-0429567, CCF-044167), and Lockheed Martin. Thanks to Curtis Clifton, David Cok, Suresh Jagannathan, Joe Kiniry, Rustan Leino, Todd Wallentine and the ECOOP program committee for comments on earlier drafts.

References

1. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (STTT) (2004) To appear.
2. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06y, Iowa State University, Department of Computer Science (2004) See www.jmlspecs.org.
3. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. Science of Computer Programming **55** (2005) 185–208
4. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D.R., Kiniry, J.: Jml reference manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org` (2005)
5. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices. (2004) To appear.
6. Meyer, B.: Object-oriented Software Construction. Second edn. Prentice Hall, New York, NY (1997)
7. SAnToS: SpEx Website. |http://spex.projects.cis.ksu.edu— (2003)
8. Robby, Rodríguez, E., Dwyer, M., Hatcliff, J.: Checking strong specifications using an extensible software model checking framework. In: Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Volume 2988 of Lecture Notes in Computer Science., Springer (2004) 404–420
9. Hatcliff, J., Robby, Dwyer, M.: Verifying atomicity specifications for concurrent object oriented software using model checking. In: Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation. Volume 2937 of Lecture Notes in Computer Science., Springer (2004) 175–190
10. Pugh, W.: Fixing the java memory model. In: Proceedings of the ACM 1999 Conference on Java Grande, New York, NY, USA, ACM Press (1999) 89–98
11. Flanagan, C., Freund, S.N.: Type-based race detection for java. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, New York, NY, USA, ACM Press (2000) 219–232
12. Lea, D.: Concurrent Programming in Java: Second Edition. Addison-Wesley (2000)
13. Raghavan, A.D., Leavens, G.T.: Desugaring JML method specifications. Technical Report 00-03d, Iowa State University, Department of Computer Science (2003)
14. Lerner, R.A.: Specifying Objects of Concurrent Systems. PhD thesis, School of Computer Science, Carnegie Mellon University (1991) TR CMU–CS–91–131.
15. Boyland, J., Noble, J., Retert, W.: Capabilities for sharing. In: Proceedings of the 15th European Conference on Object Oriented Programming. Volume 2072 of Lecture Notes in Computer Science., Springer-Verlag (2001) 1–27
16. Noble, J., Vitek, J., Potter, J.: Flexible alias protection. In: Proceedings of the 12th European Conference on Object Oriented Programming. Volume 1445 of Lecture Notes in Computer Science., Springer-Verlag (1998) 158–185
17. Müller, P., Poetzsch-Heffter, A.: Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen (2001) Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.
18. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. IEEE Transactions on Software Engineering **21** (1995) 785–798

19. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Communications of the ACM **18** (1975) 717–721
20. Flanagan, C., Qadeer, S.: Types for atomicity. In: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, ACM Press (2003) 1–12
21. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (2000)
22. Dwyer, M.B., Hatcliff, J., Robby, R.Prasad, V.: Exploiting object escape and locking information in partial order reduction for concurrent object-oriented programs. Formal Methods in System Design **25** (2004) 199–240
23. Dietl, W., Müller, P.: Universes: Lightweight ownership for jml. Journal of Object Technology (2005) To appear.
24. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, ACM Press (2003) 338–349
25. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (2004) 256–267
26. Saltzer, J.H., Reed, D.P., Clark, D.D.: End-to-end arguments in system design. ACM Transactions on Computer Systems **2** (1984) 277–288
27. Flanagan, C.: Verifying commit-atomicity using model-checking. In: Proceedings of the 11th International SPIN Workshop on Model Checking of Software. Volume 2989 of Lecture Notes in Computer Science., Springer (2004) 252–266
28. Robby, Dwyer, M.B., Hatcliff, J.: Bogor: An extensible and highly-modular model checking framework. In: Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering. Volume 28 number 5 of SIGSOFT Softw. Eng. Notes., ACM Press (2003) 267–276
29. Hartley, S.: Concurrent Programming - The Java Programming Language. Oxford University Press (1998)
30. Jacobs, B., Leino, K.R.M., Schulte, W.: Verification of multithreaded object-oriented programs with invariants. In: Proceedings of The ACM SIGSOFT Workshop on Specification and Verification of Component Based Systems, ACM Press (2004) To appear.
31. Ábrahám, E., de Boer, F.S., de Roever, W.P., Steffen, M.: A tool-supported proof system for multithreaded java. In: Proceedings of the International Symposia on Formal Methods for Components and Objects. Volume 2852 of Lecture Notes in Computer Science., Springer (2002) 1–32
32. Freund, S.N., Qadeer, S.: Checking concise specifications for multithreaded software. Journal of Object Technology **3** (2004) 81–101
33. Wang, L., Stoller, S.D.: Run-time analysis for atomicity. In: Proceedings of the Third Workshop on Runtime Verification (RV). Volume 89(2) of Electronic Notes in Theoretical Computer Science., Elsevier (2003)
34. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: Proceedings of the Second Workshop on Formal Methods in Software Practice. (1998) 7–15
35. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Robby: Expressing checkable properties of dynamic systems: The Bandera Specification Language. International Journal on Software Tools for Technology Transfer **4** (2002) 34–56

36. Deng, X., Dwyer, M.B., Hatcliff, J., Mizuno, M.: Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), New York, NY, USA, ACM Press (2002) 442–452
37. Cheon, Y., Leavens, G.T.: A runtime assertion checker for the Java Modeling Language (JML). In: Proceedings of The International Conference on Software Engineering Research and Practice, CSREA Press (June 2002) 322–328
38. SAnToS: JMLEclipse Website. |http://jmleclipse.projects.cis.ksu.edu— (2004)
39. Burdy, L., Requet, A., Lanet, J.L.: Java applet correctness: A developer-oriented approach. In: Proceedings of the 12th International Symposium of Formal Methods Europe. Volume 2805 of Lecture Notes in Computer Science., Springer-Verlag (2003) 422–439