# SAnToS Laboratory

Laboratory for **S**pecification, **An**alysis and **T**ransformation **o**f **S**oftware

Department of Computing and Information Sciences
Kansas State University

Technical Report SAnToS-TR2004-7

Last Updated: May 6, 2005

**Software Tools for Technology Transfer manuscript No.**
(will be inserted by the editor)

# Checking JML Specifications Using An Extensible Software Model Checking Framework[⋆]

**Robby**[1], **Edwin Rodríguez**[1], **Matthew B. Dwyer**[2], **John Hatcliff**[1]

[1] Department of Computing and Information Sciences, Kansas State University[⋆⋆]
e-mail: {`robby,edwin,hatcliff`}`@cis.ksu.edu`

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

May 5, 2005

**Abstract.** The use of assertions to express correctness properties of programs is growing in practice. Assertions provide a form of lightweight checkable specification that can be very effective in finding defects in programs and in guiding developers to the cause of a problem. A wide variety of assertion languages and associated validation techniques have been developed, but run-time monitoring is commonly thought to be the only practical solution.

In this paper, we describe how specifications written in the Java Modeling Language (JML), a general purpose behavioral specification and assertional language for Java, can be validated using a customized model checker built on top of the Bogor model checking framework. Our experience illustrates the need for customized state-space representations and reduction strategies in model checking frameworks in order to effectively check the kind of strong behavioral specifications that can be written in JML. We discuss the advantages and trade-offs of model checking relative to other specification validation techniques and present data that suggest that the cost of model checking strong specifications is practical for several real programs.

[⋆] This is an extended version of the paper *Checking Strong Specifications Using An Extensible Model Checking Framework* that appeared in *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2004*. This work was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by NSF (CCR-0306607) by Lockheed Martin, and by Rockwell-Collins.

[⋆⋆] 234 Nichols Hall, Manhattan, KS 66506, USA.

[⋆⋆⋆] 256 Avery Hall, Lincoln, NE 68588, USA.

## 1 Introduction

The idea of interspersing specifications of the intended behavior of a program directly in the source code is nearly as old as programming itself [11]. Those foundational ideas inspired the development of more elaborate design practices and methodologies, for example, design-by-contract [22]. The use of assertional specifications has long been regarded as a means for improving software quality, but only recently have studies demonstrated support for this conjecture [32]. The increasing number of modern languages (e.g., Java, C#, PHP) and implementation frameworks (e.g., Microsoft Foundation Classes (MFC)) that include simple assertion mechanisms suggests that they will finally have the practical impact that was predicted decades ago.

To fulfill this promise, there is a need for program assertion checking mechanisms that are cost-effective, automatic, and thorough in considering both specification and program behavior. Run-time monitoring of assertions during program execution is the only mechanism that is widely used in practice today. It is both cost-effective and automatic, but can only analyze the program behaviors that are actually executed. This lack of coverage of program behavior is a significant weakness of run-time methods, especially for concurrent programs where subtle errors may depend on the order in which threads execute. To address this behavior coverage problem, a variety of static analysis approaches have been proposed to thoroughly check a program's possible behaviors with respect to certain lightweight specifications, such as pointer non-nullness and array bounds [10], and propositional temporal properties [39]. These methods gain program coverage by sacrificing the expressiveness of their specification language.

Building on a long line of work on formal methods for manual reasoning about complete behavioral specifications of programs, several recent languages have emerged that balance the desire for completeness and the pragmatics of checkability. The Java Modeling Language (JML) is one such language [20]. With JML, one can specify properties of varying strength from lightweight assertions about pointer null-ness to complete functional correctness of program components; the latter we refer to as a *strong* property. 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 [22] paradigm by including notation for pre/postconditions and invariants. JML uses Java's expression syntax and adds constructs that dramatically increase expressiveness (e.g., it is possible to quantify, universally or existentially, over objects in the heap).

In this paper, we describe how we have adapted a flexible model checking framework called Bogor [28] to check JML specifications of sequential and concurrent Java programs. Model checking adds a new and complementary approach to the existing run-time and theorem-proving technologies for reasoning about JML. While tools based on those technologies have proven effective in supporting certain kinds of Java validation and verification activities, there is currently no *automatic* technique for *thoroughly* checking a wide-range of *strong* JML specifications especially in the presence of *concurrency*. Our checking tool is automatic and exhaustive in its reasoning about general JML properties up to user defined bounds on the space consumed by a program run.

Using existing model checking techniques to verify strong specifications is problematic for several reasons. First, existing model checkers, such as Spin [16], do not provide direct support for modeling dynamically allocated objects and heap structures making it difficult to represent the program's behavior; Bogor maintains an explicit, yet compact, representation of the dynamic program heap [29]. Second, even if one could encode the behavior in the input language of such a model checker, the underlying checking algorithms would not exploit the semantic properties of the original language to optimize the state space search; Bogor incorporates novel partial order reductions that exploit the semantics of a program's heap and locking structure to achieve efficiency [6]. Finally, existing model checking frameworks support temporal properties but do not provide direct support for expressing rich data or heap-related functional properties; Bogor supports extension of the expressions sublanguage via user defined atomic expressions that can be evaluated over the full extent of a program state including the heap [28].

The contributions of this paper are as follows:

– we demonstrate that with a sufficiently feature-rich model checking framework one can check strong behavioral specifications;
– we describe how Bogor's extension facilities can be applied to implement checking of JML specifications, including specifications that have proven difficult to check by other means such as run-time checking or theorem-proving; and
– we demonstrate that the overhead of checking JML specifications can be mitigated, and in most cases completely eliminated, through the use of sophisticated state-space reductions.

In the next section, we give an overview of JML and illustrate its main features through an example. Section 3 describes the features of the Bogor model checking framework that enable the efficient treatment of JML specifications. There are a number of different techniques for reasoning about JML specifications; we compare representatives of the main classes of techniques to our model checking approach in Section 6. Section 4 details our strategy for efficiently reasoning about JML specifications on-the-fly during state-space exploration of a concurrent Java program. In Section 5, we detail the analysis of a collection of JML annotated Java programs and report on the cost and effectiveness of checking them with Bogor and then conclude. We also refer the reader to the SpEx web site [33] for the complete JML-annotated Java code and the associated Bogor models for all of the examples considered in this paper.

## 2 JML: The Java Modeling Language

The Java Modeling Language (JML) [20] is a Java-specific behavioral specification language [40] designed at Iowa State University by Gary Leavens and others. We illustrate JML and the treatment of JML specifications with the example in Figure 1. This example is a concurrent linked-list-based queue from [19] with some JML specifications, written in Java comments with special tags such as `//@`, added to describe its behavior.

Instances of the class `LinkedNode` implement the nodes of the linked list representing the queue. The `LinkedQueue` class provides `put` and `take` methods that implement a fine-grained locking protocol, through the use of the protected methods `insert` and `extract`, to maximize concurrent access to the queue. This design leads to functional code that is nested inside synchronized statements and conditionals in those protected methods. In order to specify the behavior of that functional code we have refactored them into additional protected methods, e.g., `refactoredInsert`.

When a new queue is created, an object that is used to guarantee mutual exclusion of `put` operations is created and assigned to the `putLock` field and a new node is created and assigned to the `head` and `tail` instance fields (this *dummy* node, with an unused data field, forms the head of every list). Whenever a thread attempts to `take` an object from an empty

```
class LinkedNode {
 public Object value;
 public LinkedNode next;

 /*@ behavior ensures value == x &&
   @                    next == null;
   @*/
 public LinkedNode(Object x) {
   value = x;
 }
 ...
}

public class LinkedQueue {
 protected final /*@ non_null @*/ Object putLock;
 protected /*@ non_null @*/ LinkedNode head;
 protected /*@ non_null @*/ LinkedNode last;
 protected int waitingForTake = 0;

 ...
 //@ instance invariant waitingForTake >= 0;
 //@ instance invariant \reach(head).has(last);

 /*@ behavior
   @   assignable head, last, putLock, waitingForTake;
   @   ensures \fresh(head, putLock) &&
   @           head.next == null;
   @*/
 public LinkedQueue() {
  putLock = new Object();
  last = head = new LinkedNode(null);
 }

 /*@ behavior
   @   ensures \result <==> head.next == null;
   @*/
 public boolean isEmpty() {
  synchronized (head) {
   return head.next == null;
  }
 }

 /*@ behavior
   @    requires n != null;
   @    assignable last, last.next;
   @*/
 protected void refactoredInsert(LinkedNode n) {
  last.next = n;
  last = n;
 }
/*@ behavior
  @  requires x != null;
  @  ensures true;
  @ also behavior
  @  requires x == null;
  @  signals (Exception e)
  @   e instanceof IllegalArgumentException;
  @*/
public void put(Object x) {
 if (x == null)
  throw new IllegalArgumentException();
 insert(x);
}

protected synchronized Object extract() {
 synchronized (head) {
  return refactoredExtract();
 }
}
/*@ behavior
  @  assignable head, head.next.value;
  @  ensures \result == null || (\exists LinkedNode n;
  @          \old(\reach(head)).has(n);
  @                n.value == \result
  @                && !(\reach(head).has(n)));
  @*/
protected Object refactoredExtract() {
 Object x = null;
 LinkedNode first = head.next;
 if (first != null) {
  x = first.value;
  first.value = null;
  head = first;
 }
 return x;
}

/*@ behavior
  @  requires x != null;
  @  ensures last.value == x && \fresh(last);
  @*/
protected void insert(Object x) {
 synchronized (putLock) {
  LinkedNode p = new LinkedNode(x);
  synchronized (last) refactoredInsert(p);
  if (waitingForTake > 0) putLock.notify();
  return;
 }
}

//@ ensures \result != null;
public Object take() {
 Object x = extract();
 if (x != null) return x;
 else {
  synchronized (putLock) {
   try {
    ++waitingForTake;
    for (;;) {
     x = extract();
     if (x != null) {
      --waitingForTake;
      return x;
     }
     else putLock.wait();
    }
   }
   catch (InterruptedException ex) {
    --waitingForTake;
    putLock.notify();
    throw new RuntimeException();
   }
  }
 }
}
```

**Fig. 1.** A Concurrent Linked-list-based Queue Example (excerpts)

queue, the thread is blocked. If the queue is not empty, then only the head is locked, and its stored value is returned. The dequeuing is done in the `extract` method. Whenever an object is enqueued, the tail is locked, a new node is created to store the object and one of the threads waiting to dequeue is notified.

JML specifications are phrased as invariants on instances of classes and contracts for method invocations. One important aspect of JML is that it balances support for complete behavioral specification with lightweight assertion features, such as Java assertions. Thus, it allows developers to vary the *strength* of their specifications across classes and within classes. The variation in the strength of specifications in the example is apparent (e.g., method `extract` has no specification whereas method `refactoredExtract` has a behavioral specification that captures removal of an element from the queue).

Invariants on objects are stated using `invariant` clauses. In an object-oriented language such as Java, the notion of an object invariant is different than the traditional model checking definition of invariant, which requires a condition to hold at every reachable system state. The intuition is that an object invariant is not required to hold before the object is initialized nor during execution of methods acting upon the object. JML invariants for instances of a class $C$ are required to hold in *visible states* which are defined at the end of execution of a constructor for $C$ and at the entry and exit of method calls. This last condition ensures that changes to a public field that do not occur through methods of $C$ are visible to the invariant. In the example, the first invariant in class `LinkedQueue` states that the integer field `waitingForTake` is non-negative. For convenience several short-hand type modifiers, such as `non_null` which specifies that a reference field never has a null value, are also defined. Class `LinkedQueue` specifies that all of its reference fields are non-null using this JML modifier.

Method preconditions are written using the JML `requires` clause, which specifies the conditions on program state when the method is called (i.e., the method *prestate*) that callers must assure. Postconditions for non-exceptional methods are written using the `ensures` clause, which specifies the conditions on the program state upon return (i.e., the method *poststate*) that the method guarantees. JML's `invariant`, `requires`, and `ensures` clauses specify a boolean condition on the state of the program that can be expressed using Java's expression syntax and a set of JML operators including:

– `\old`($e$): used only in postconditions to access the value of expression $e$ in the method prestate.
– `\reach`($e$): returns the set of objects that are reachable through chains of field accesses from reference $e$.
– `\forall`, `\exists`: allows one to state properties using logical quantification; this is especially useful for stating properties about all or some of the objects in the heap.
– `\fresh`($x_1, \ldots, x_n$): used only in postconditions to state that the variables $x_i$ hold objects not allocated in the prestate.
– `\result`: evaluates to the return value of a method.

These operators allow for strong behavioral properties to be specified. For example, the `refactoredExtract` method's postcondition states that the result is either null (when the list is empty) or that there exists a node $n$ such that $n$ is in the list in the prestate of the method, $n$'s `value` is returned as the method result, and $n$ is not in the list in the poststate of the method. To express this intent the specification captures the set of objects on the heap that are reachable from the `LinkedQueue` head, using `\reach`, in the method prestate, using `\old`, and then quantifies over them on method exit, using `\exists`. As another example, the `insert` method has a precondition that requires its argument, giving the object to be inserted, to be non-null, and a postcondition that ensures the last node in the list is newly allocated in the call, using `\fresh`, and holds the object supplied for insertion.

The `assignable` clause states a form of *frame condition* for a method: only reference expressions that are listed in `assignable` may be modified by a method call. Locations that are local to the method (or any methods that it calls) and locations that are created during the method's execution are not subject to this restriction. This clause is used in the `refactoredExtract` method to state the invariance of the `LinkedQueue` head and the initial dummy node across the call.

Syntactically, a specification is heavyweight as long as it is annotated using one of the `behavior` keywords, otherwise the specification is lightweight. The `put` method illustrates a *heavyweight* specification where the possible invocation states are covered by a pair of `behavior` specifications consisting of `ensures` and `signals` clauses; JML provides the `signals` clause to specify the conditions under which a method returns via exception throw. As complete behavioral specifications, heavyweight JML specifications always have a well-defined default value for any omitted clause. Table 1 defines the default values for missing clauses in both heavyweight and lightweight JML specifications; the special JML tag `\not_specified` means that the clause is free to take on any legal value.

JML is a large and complex specification language. Rather than provide a detailed description of all of its language fea-

| Omitted Clause | Lightweight | Heavyweight |
|---|---|---|
| `requires` | `\not_specified` | `true` |
| `diverges` | `\not_specified` | `false` |
| `assignable` | `\not_specified` | `\everything` |
| `ensures` | `\not_specified` | `true` |
| `signals` | `(Exception) \not_specified` | `(Exception) true` |

**Table 1.** Default values for JML clauses in lightweight and heavyweight specifications.

tures, in the subsequent presentation, we focus on those features that are problematic to check with existing technologies or that raised particular issues in the implementation of our model checking support. A complete discussion of our support for JML features is given at [33].

## 3 The Bogor Model Checker

Bogor [28] is an extensible and highly modular software model checking framework that can be adapted and customized to the specific characteristics of different problem domains. Figure 2 depicts the internal architecture of Bogor. What makes Bogor unique is that it is designed to be easily customized both in terms of its input language, through the use of front-end and interpretive components, and in terms of the semantics of the modules that implement the core capabilities of the explicit-state model checking algorithms, depicted on the right side of Figure 2. For example, Bogor has been customized to analyze properties of real-time embedded design models of avionics systems [7] by both adding primitives for modeling the semantics of avionics middleware services and by customizing the state-storage and search strategies of the model checker based on those semantics. In this section, we present an overview of the extension mechanisms in Bogor since they are the vehicle through which JML specifications are checked. We also survey several customized Bogor modules that exploit the semantics of JVM byte-codes to improve the performance of model checking multi-threaded Java programs. Those modules provide essential capabilities for encoding JML operations and for eliminating the overhead of checking JML specifications against Java code.

### *3.1 Language Extension*

Like many model checkers Bogor's input language BIR, an example of which is given in Figure 3, is a guarded command transition system language with explicit locations (`loc`), explicit guards (`when`), sequences of statements comprising a transition's action (`do`), and explicit transitions (`goto`). Unlike most model checkers, BIR includes support for dynamically allocated data (`record`, `new`), method calls (`function`), and other features needed to model JVM byte-codes, such as exceptions. In addition, BIR features an extension language facility that allows introductions of new Abstract Data Types (ADTs) and abstract operations as first-class constructs of the language. These introductions are analogous to adding new native types and native instructions, so they can be used to build abstract machines tailored to specific application domains.

The figure presents a BIR model for a simple system in which there are several processes competing for resources collected in a pool. For this particular system, we only care about membership operations on the resource pool, thus, we create a new *Set* ADT to model this pool of resources. To do this, the keyword `extension` is used to declare the extension name. The `for` keyword provides the fully qualified name of the class that implements the extension. The idea is that by introducing a customized extension, the concept of a *set* data structure is made a primitive component of the language, allowing: (*i*) hiding all the implementation complexity from the model and (*ii*) the introduction of optimizations based on the state of the set. The front-end components support extension syntax directly and do not require the user to modify any internal Bogor components. Extensions map the newly introduced type to instances of Bogor interpretive interfaces and implement the semantics of extension methods, such as `add<'a>`, as Java that manipulates those structures. These implementations follow a very regular pattern and use a clean and well-defined API using widely known design patterns [12], thus making it easy for non-model checking experts to extend the toolset.

We define a BIR extension for each JML construct and embed the checks for a JML specification directly into the BIR representation that is compiled from Java source code. We note that encoding JML constructs directly in the input language of most existing model checkers would be difficult or impossible in many cases. Features like `\reach`, which

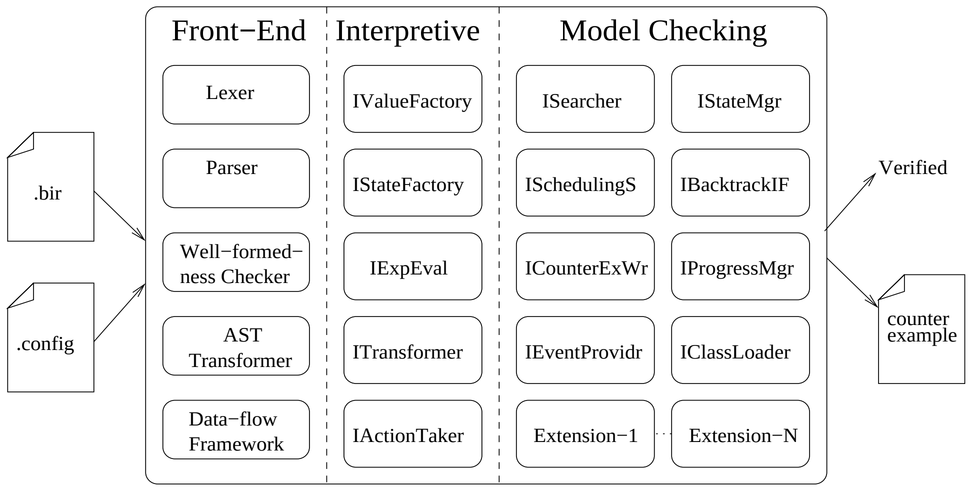

**Fig. 2.** Bogor Architecture.

```
system ResourceContention {
  extension Set for myPackage.SetModule {
    typedef type<'a>;
    expdef Set.type<'a> create<'a>('a ...);
    expdef 'a choose<'a>(Set.type<'a>);
    expdef boolean isEmpty<'a>(Set.type<'a>);
    expdef boolean forAll<'a>('a -> boolean,
                              Set.type<'a>);
    actiondef add<'a>(Set.type<'a>, 'a);
    actiondef remove<'a>(Set.type<'a>, 'a);
  }

  record Resource { boolean isFree; }
  record Disk extends Resource { }
  record Display extends Resource { }

  Set.type<Resource> resourcePool;

  fun isResourceFree(Resource resource)
      returns boolean = resource.isFree;

  fun AreAllResourcesInPoolFreeInv()
      returns boolean =
    Set.forAll<Resource>(isResourceFree,
                         resourcePool);

  main thread MAIN() {
    loc loc0:
      do { // create the pool and creates two processes
        resourcePool := Set.create<Resource>
            (new Disk, new Disk, new Display);
        start Process(); start Process();
      } return;
  }

  thread Process() {
    loc loc1:
      invoke run()
      return;
  }

  function run() {
    Resource resource;
    loc loc2:
      when !Set.isEmpty<Resource>(resources)
      do { // choose an element and remove it
        resource := Set.choose<Resource>
            (resourcePool);
        Set.remove<Resource>(resourcePool,
                             resource);
      } goto loc3;
    loc loc3:
      do { // resource in use
        resource.isFree := false;
      } goto loc4;
    loc loc4:
      do { // resource free
        resource.isFree := true;
      } goto loc5;
    loc loc5:
      do { // add the resource back to pool
        Set.add<Resource>(resourcePool,
                          resource);
      } goto loc2;
      do { // empty transformation
      } goto loc2;
  }
}
```

**Fig. 3.** Resource Contention Example

requires scanning the heap for reachable sets of objects, would require an iterative or recursive block of code to be embedded into the model checker input. Bogor extensions are implemented in Java (in the model checker rather than in the input model) which alleviates this problem.

### *3.2 Module Extension*

Bogor can be thought of as a well-factored design for an explicit-state model checker that is encoded into the module structure of a Java implementation. Figure 2 shows the functionality common to every explicit-state model checker, for example, the storage of system states (`IStateMgr`), the particular form of search implemented (`ISearcher`), the strategy used to select the next transition to explore (`ISchedulingStg`) and information needed to guide backtracking in the search (`IBackTrackIF`). These interfaces form the *extension points* for the core Bogor model checking framework. Developers can implement custom versions of components that implement the interfaces and then configure Bogor to use selected components. This greatly simplifies customization of the model checker.

To implement certain JML operators, such as `\old`, one needs to inspect the state of the heap and access information about method prestates. This functionality cannot be implemented in a BIR language extension alone, but it can be achieved through Bogor module extension. We present the details of JML module extensions in Section 4, but to set the stage for that explanation we provide an overview of several Java-specific module extensions.

**Heap and Thread Symmetry** Bogor exploits similarity in the behavior of threads and in the structure of the program heap to define equivalence classes of states.

The semantics of Java dictate that a program can observe neither the physical address of an object nor the unreclaimed unreachable objects (i.e., garbage). For these reasons, execution states of a Java program that differ only in the physical addresses of objects or in the unreclaimed garbage can be considered equivalent. Therefore, any property that holds for a state also holds for any equivalent state. It is common in Java programs for multiple instances of a thread type to be running at a given time. Such threads are distinguishable only by their thread object references which can only be observed by explicit reference equality comparisons. For programs without such comparisons, states that differ only in the identity of the threads that are at specific points in their execution can be considered equivalent. Using these observations, the time and space required for model checking a system can often be reduced by restricting state-space search to a single representative of each equivalence class.

Bogor's heap symmetry reduction is implemented in a custom `IStateMgr` module extension that topologically sorts the heap based on a lexicographic ordering of the reference chains that reach each object in the heap [29]. Since this module already traverses the heap, it is natural to implement JML operators like `\reach` as a variation on this module that restricts itself to a portion of the heap rooted at a given reference.

**Collapse Compression** Bogor reduces the space required to store a state by sharing common parts of distinct states. This technique leverages the fact that a transition usually only modifies a small part of the state; in Java model checking, a transition models a JVM byte-code which typically modifies a single variable or object field.

Most explicit-state model checkers [16,2] store states in a compressed form while maintaining a small number of uncompressed state instances for direct manipulation during transition execution. A compressed state, which is encoded as a bit-vector, acts as a unique *fingerprint* for the state that is used to determine whether the state has been encountered previously during search. Bogor's collapse compression exploits the fact that a Java program's state can be represented as a hierarchy of components (e.g., each thread has separate locals and control information, the topologically sorted heap is a hierarchy of sub-heaps) [29]. This makes it possible to localize the effect of a transition on the overall state. For example, a transition that updates a thread's local variable is guaranteed to preserve the other thread's data and the heap. When compressing a state, Bogor reuses the portion of the previous state's fingerprint that corresponds to the unchanged portions of the state; this significantly reduces the cost of state compression and the space required to represent the state.

Accessing information about the history of a trace in a stateful search requires that traces be distinguished by different history values. Since JML operators, like `\old`, reference method prestate information, correct stateful search of JML annotated Java code must distinguish traces based on prestates. Efficient state-compression techniques are useful for encoding portions of method prestates that are referred to in JML postconditions and the resulting fingerprints are an efficient way of distinguishing traces leading to those postconditions.

**Backtracking Depth-first Search** Bogor performs a depth-first search and, like Spin, it is designed to backtrack by executing *undo actions* for previously taken transitions. This eliminates the need to uncompress states which can be expensive. For implementing JML operators that access history information, such as `\old`, it has the negative consequence of not allowing access to the values of a state given its fingerprint. In Section 4, we describe how a custom version of the backtracking module is used to walk back through the transitions in a method body to construct the portion of the prestate that is referenced in the postcondition by an `\old` expression.

**Partial Order Reductions** Bogor minimizes the set of paths that need to be explored in the state-space during model checking. Classical partial order reductions (POR) (e.g., [5]) leverage the *independence* of transitions to induce equivalence classes of paths such that it is sufficient to explore a single path from each class. Intuitively, a pair of transitions are independent if the execution of one transition cannot influence the execution of the other. Existing implementations of partial-order reductions (e.g., SPIN) use efficient techniques for safely approximating the set of independent transitions. For example, transitions that access only thread-local variables cannot influence transitions in other threads and can therefore be classified as independent of all other transitions by a simple syntactic analysis.

For multi-threaded Java programs, we leverage the structure of the Java heap and Java synchronization idioms to infer more precise information about transition independence [6]. For example, transitions that access *thread-local* objects (i.e., objects that are reachable from a single thread) are independent because no transition in another thread can possibly access such an object (until the object becomes shared). Transitions that operate on a *properly locked* object (e.g, where the set of locks held by each thread when accessing the object always contains at least one common lock) are also independent as are operations on *read-only* objects. An additional feature of Bogor's POR implementation is that it biases the search to coalesce transitions in a method into a consecutive run of transitions. By doing this, the model checker is able to defer state-storage until the end of the run of independent transitions effectively implementing on-the-fly detection of *atomic blocks* of transitions.

The combination of partial order reduction techniques in Bogor yields orders of magnitude reduction in the space and time required for model checking nearly all of the Java programs we have encountered. Furthermore these reductions prove very useful when checking JML specifications. If an entire method execution can be treated as an atomic block, then even if the postcondition of that method accesses all of the prestate of the method there is no additional state-storage required. In such situations, the overhead of complex JML operators such as `\old` can be completely eliminated by state-of-the-art POR implementations.

## 4 Checking JML Specifications with Bogor

### 4.1 Implementation Strategy

All existing JML checking tools of which we are aware of employ a two-phase implementation strategy. In the first phase, JML specifications along with the associated Java code are translated to a lower-level representation. In the second phase, the lower-level representations are checked using the corresponding verification technologies.

It is important to note that a significant portion of the effort in implementing JML checking is associated with the translation phase. Implementation of the translation phase is non-trivial, since it is this phase that captures JML semantics of specifications associated with class inheritance, method overriding, etc. [20]. For example, the "effective precondition" (i.e., the condition that should actually be checked as compared to the one that is written in JML comments) of a method that overrides previously defined methods, is a combination of all the preconditions listed in the current method conjoined with all preconditions defined in the method of the same signature above the present one in the inheritance hierarchy. Specifications for implemented interfaces must also be taken into account (e.g., by combining the pre/postconditions declared on methods in interfaces with the pre/postconditions of the implementing methods). In addition, since invariants are checked at method entry/exit, invariants are conjoined with pre/postconditions to form the effective pre/postconditions.

To avoid much of the effort in implementing appropriate pre/postconditions in the translation phase described above, we have chosen to reuse part of the (`jmlc`) JML run-time checker infrastructure developed by Cheon and Leavens [4]. `jmlc` translates Java source code annotated with JML specifications into byte-code that includes run-time assertions that encode the specifications. For our Bogor JML checking infrastructure, we are in the process of modifying the JML translation module of the `jmlc` compiler to target BIR plus the BIR language extensions that we introduce in the remainder of this section to support specific JML constructs [1]. This Java/JML translation results in a representation in which BIR code that realizes system semantics and BIR code that realizes program specifications are integrated into a single model. This combination of system and specification representations occurs not only in `jmlc`, where Java byte code represents both system behavior and assertions representing specifications, but also in theorem-proving-based JML tools such as

[1] The specifications used in this paper were translated manually according to the approach taken by `jmlc`, for evaluation purposes, in Section 5.

ESC-Java [10] and LOOP [38], where logic formulas represent both system behavior and specifications.

In a representation that combines system and specifications using an executable representation like BIR or byte code, one may have concerns that execution of code representing system behavior may interfere with the execution of code representing specifications. When multi-threaded programs are considered, this is actually a problem in runtime-monitoring approaches like `jmlc` as we explain in greater detail below. Thus, one might imagine an approach for checking JML with a model checking engine in which specifications are not inserted directly into the system BIR, but instead simple *triggering* events are inserted that cause the checking of specifications stored separately from the system BIR (i.e., stop the state-space exploration, check the specifications, and resume). In fact, Bogor conceptually achieves this form of separation due to the fact that all system execution is suspended (i.e., not considered for scheduling) when specifications are being executed. Moreover, as we explain in Section 4.10, Bogor checks that specifications are *pure* in the sense that their evaluation does not produce any visible changes to the state. Thus, evaluation of the BIR representing specifications will not interfere with the current state to be used in evaluating the BIR representing the system.

The key to our implementation strategy is that Bogor is, in essence, an extensible interpreter in which rich verification primitives (e.g., quantification over heap structures) can be implemented directly using Bogor's modeling language extension primitives and in which direct control over action execution (e.g., scheduling of thread actions) can be obtained using Bogor's pluggable state-space exploration engine modules. To emphasize this point, below we summarize the principles of our implementation strategy in which the flexibility of Bogor is used to implement the verification of a rich subset of the JML language. In particular, we seek to draw contrasts with the checking approach of `jmlc` which most closely matches the architecture of our model-checking-based solution. The contrasts that we draw with `jmlc` stem from the fact that Bogor provides significant flexibility in terms of *granularity* of actions as well as *control* over when those actions are scheduled for evaluation in relationship to actions from other threads. The rest of Section 4 presents a detailed discussed of how these principles were applied for particular JML constructs.

**Rich verification primitives:** `jmlc` must represent all verification requirements as regular Java bytecode which has a *fixed granularity*. With Bogor, we add primitives to the modeling language to directly represent almost all JML constructs such as quantifications, `\reach`, `\old`, etc. Many of these constructs are difficult to represent using Java bytecode/assertions. For example, the general form of universal quantification in `jmlc` involves instrumenting the Java code to build extra data structures that hold references to all allocated objects of a particular type. This is difficult to support without modifying the Java Virtual Machine. Realizing the semantics of these constructs inside the Bogor execution engine itself instead of in the interpreted language also reduces the *interpretive overhead* associated with executing these statements.

**Direct access to underlying data structures representing the heap:** When one adds extensions to Bogor's modeling language, the semantics of extensions is implemented by plugging in code to the Bogor interpretive engine. This code has full access to Bogor's internal representations, including its representation of the heap. Thus, constructs such as universal quantification and `\reach` are easily implemented by walking over the Bogor representation of the current state.

**Direct access to state history:** The semantics of `\old` in which the state of all objects reachable from the argument of `\old` must be preserved is virtually impossible to implement using only bytecode/assertions without modifying the Java Virtual Machine (e.g., not all Java classes are cloneable, comparison between cloned objects is problematic due to mismatches in reference values, etc.). Since Bogor stores history information, the task of implementing `\old` in Bogor is made easier by calling the state history management facilities in Bogor to re-create relevant portions of a method's prestate.

**Control of interleaving:** In the presence of concurrency, it is difficult to apply the `jmlc` approach to implementing checking of JML pre/post conditions or invariants using bytecode/assertions since, conceptually, evaluations of these expressions should happen in a single atomic step. One might imagine using Java's locking mechanism to avoid interference associated with interleaving of other thread actions during the actions of specification checks. However, there are a number of problems with locking individual objects occurring in the expressions (e.g, undesirable interference can still occur unless all the objects are locked in a single step). In Bogor, since extension implementations have complete control of the Bogor scheduler, other threads can simply be suspended during the evaluation of a specification expression – which effectively allows the expression to be evaluated in a single atomic step in relation to other thread actions. Furthermore, it is the direct control of interleaving that allows the model checking engine to explore all possible schedules for the program – unlike the relatively low coverage of concurrent interleavings obtained by run-time monitoring.

### *4.2 Translating JML to BIR*

In this section, we explain the details of how JML annotations are translated to BIR. Space constraints only allow a brief overview of the translation. We refer the reader to [33] for complete JML-annotated source and the BIR code that results from the translation for all the examples listed in the experimental studies of Section 5.

Figure 4 shows the translation to BIR code of the method `refactoredExtract()` of the linked queue program in Figure 1. The Java program is translated by a tool called J2B (Java to BIR) and it is equivalent, instruction by instruction, to the original program, that is, there is no abstraction involved. We will be using this figure throughout the next sections to describe the implementation of several JML features. Figure 4 displays only the specification code. These are the instructions that are added to the generated BIR code to check JML specifications; the instructions corresponding to the body of the method have been elided.

Most of the JML annotations are translated to BIR assertions. Some others are translated to specific operators that have been added as BIR extensions that realize the JML checking algorithm. The assertions are normally inserted in every method's pre and poststate, because these are, according to JML definition, the visible states.

To ensure proper behavior, assertions that correspond to the same group of checks in the method (e.g. corresponding to a postcondition, an invariant, etc.) are all grouped together in a single atomic block. This ensures that the execution of specification assertions is totally invisible and separated from the actual system code and does not interfere with the underlying model. This is true because no system instructions are executed while a JML specification is checked and because JML checks leave the system in the same state as it was before the check was performed. For example, we can see in Figure 4 how all the checks for the precondition and invariants have been grouped in a single location: `locSpec1`.

Further details about the verification code in Figure 4, with their relationship to the corresponding JML annotations, will be given throughout the remaining subsections (4.3 to 4.4).

### *4.3 Lightweight versus Heavyweight Specifications*

As explained in Section 2, JML provides two different styles of specifications, based on their thoroughness: lightweight and heavyweight specifications. Our framework checks both types of specification in a straightforward manner because most specifications are translated to simple assertions (with the exception of invariants as described in Section 4.7). Lightweight specifications, which are already assertions, are simply inserted in the appropriate position in the BIR code, whereas the heavyweight specifications, which are usually much more complex since they involve multiple behavior clauses, are translated to a sequence of assertions in the BIR code.

### *4.4 Logic Operations*

In this section, we discuss how logic operations are handled in Bogor. The conventional logic operators of conjunction (&&), disjunction (`||`), negation (`!`), etc., are built in as part of BIR syntax. However, JML has facilities to define both universal and existential quantification.

The universal quantification expression `\forall`($\tau$ $X$; $R(X)$; $C(X)$) holds true when $C(X)$ is satisfied by all values of quantified variables $X = x_1, \ldots, x_n$ of type $\tau$ that satisfy the range predicate $R(X)$. Bogor supports bounded (finite) quantifications over integer types and quantifications over reference types. Quantifications over reference types are implemented by collecting the set of reachable $\tau$ objects from all global variables and threads.

The existential quantification expression `\exists`($\tau$ $X$; $R(X)$; $C(X)$) holds true if $C(X)$ is satisfied by some values of quantified variables $X = x_1, \ldots, x_n$ of type $\tau$ that satisfy the range predicate $R(X)$. This quantification is supported similarly as `\forall` – values of the associated domain are considered in sequence until a value is found that satisfies $C(X)$.

### *4.5 Heap Object Operations*

JML provides a rich set of operators that allow the manipulation of objects stored in the heap. Bogor maintains an explicit representation of the heap and its contents. Therefore, implementing these operators in Bogor is done by an inspection of the heap representation. In the following paragraphs, we explain some of these JML operators and how they have been easily implemented in Bogor.

`\reach`($e$) gives the objects reachable by following reference chains originating from $e$. JML also includes variants of `\reach` that filter the objects based on their types and field navigations [20]. The basic notion of heap reachability is used extensively in Bogor for partial-order reductions and thread symmetry reduction as described in Section 3. Given this existing functionality in Bogor, `\reach`($e$) is easily evaluated by calling the appropriate Bogor libraries. The last as-

```
function {|linkedqueue.LinkedQueue.refactoredExtract()|}((|linkedqueue.LinkedQueue|) [|r0|])
  returns (|java.lang.Object|) {
        (|java.lang.Object|) [|r1|]; (|linkedqueue.LinkedNode|) [|$r1|];
        (|linkedqueue.LinkedNode|) [|r2|]; Set.type<(|java.lang.Object|)> spec1;
        int collapsedState;
    loc locSpec0:
      do invisible { /* nothing */ } goto locSpec1;
    loc locSpec1:
      do invisible {
        collapsedState :=
          State.getCollapsedState<(|linkedqueue.LinkedNode|)>([|r0|]./|linkedqueue.LinkedQueue.head|\);

        // Invariants
        assert([|r0|]./|linkedqueue.LinkedQueue.head|\ != null);
        assert([|r0|]./|linkedqueue.LinkedQueue.last|\ != null);
        assert([|r0|]./|linkedqueue.LinkedQueue.putLock|\ != null);
        assert([|r0|]./|linkedqueue.LinkedQueue.waitingForTake|\ >= 0);
        assert(Set.contains<(|java.lang.Object|)>(State.reachSet<(|java.lang.Object|)>([|r0|]
                ./|linkedqueue.LinkedQueue.head|\), [|r0|]./|linkedqueue.LinkedQueue.last|\));

        // Checking of Frame Conditions
        State.enterAssignable();
        State.addAssignable<(|linkedqueue.LinkedNode|)>([|r0|]./|linkedqueue.LinkedQueue.head|\);
        State.addAssignable<(|java.lang.Object|)>([|r0|]./|linkedqueue.LinkedQueue.head|\
          ./|linkedqueue.LinkedNode.next|\./|linkedqueue.LinkedNode.value|\);

        // First method's instruction...
        [|r1|] := null;
      } goto loc7;

                . . . // Instructions corresponding to the body of the method

    loc locSpec5:
      do invisible {
        // Last method's instruction
        [|r0|]./|linkedqueue.LinkedQueue.head|\ := [|r2|];

        // End of Frame Conditions Check
        State.exitAssignable();

        // postcondition
        spec1 := State.preVal<Set.type<(|java.lang.Object|)>>
                  (State.reachSet<(|java.lang.Object|)>([|r0|]./|linkedqueue.LinkedQueue.head|\),
                   State.getCurrentThreadId());
        assert([|r1|] == null || Set.exists2Context<(|java.lang.Object|), (|java.lang.Object|),
                                                      Set.type<(|java.lang.Object|)>>
                                 (specFun1, spec1, [|r1|], State.reachSet<(|java.lang.Object|)>
                                  ([|r0|]./|linkedqueue.LinkedQueue.head|\)));

        // Invariants
        assert([|r0|]./|linkedqueue.LinkedQueue.head|\ != null);
        assert([|r0|]./|linkedqueue.LinkedQueue.last|\ != null);
        assert([|r0|]./|linkedqueue.LinkedQueue.putLock|\ != null);
        assert([|r0|]./|linkedqueue.LinkedQueue.waitingForTake|\ >= 0);
        assert(Set.contains<(|java.lang.Object|)>
                (State.reachSet<(|java.lang.Object|)>([|r0|]./|linkedqueue.LinkedQueue.head|\),
                 [|r0|]./|linkedqueue.LinkedQueue.last|\));
      } goto loc13;
    loc loc13:
      do { /* nothing */ } return [|r1|];
}

fun specFun1((|java.lang.Object|) n, (|java.lang.Object|) r,
  Set.type<(|java.lang.Object|)> s) returns boolean =
        n instanceof (|linkedqueue.LinkedNode|) ?
            (((|linkedqueue.LinkedNode|))n)./|linkedqueue.LinkedNode.value|\ == r &&
                                                    !(Set.contains<(|java.lang.Object|)>(s, n)) : false;
```

**Fig. 4.** Specification check code for method `refactoredExtract()` of the Linked Queue example

sertion at `locSpec5` in Figure 4 illustrates how those library routines are exposed as BIR extensions.

`\lockset` gives all the objects locked by the current thread. The notion of lock set is already used in Bogor's partial order reductions as well, and just as with `\reach`, it can be implemented by calling the existing Bogor libraries.

### *4.6 Checking Pre and Postconditions*

The JML constructs `requires`, `ensures` and `signals` are used to specify preconditions, normal postconditions, and exceptional postconditions, respectively. Normal postconditions are checked on method exits caused by executing a `return` bytecode in Java, and exceptional postconditions are checked on exits caused by an uncaught exception. As described earlier, we check preconditions for a thread $t$ entering a method $m$ when $t$'s program counter (PC) is at the first bytecode instruction of $m$. The Bogor representation of the precondition is wrapped together with the representation of the first bytecode of the method in a Bogor atomic block, which guarantees that no interleaving can occur from the start of the checking until after the completion of the first byte code. This is illustrated in location `locSpec1` of Figure 4: all the assertions to check the precondition and class invariants are grouped in the same location, together with the first method's instruction. As mentioned earlier, all instructions in the same location are executed atomically without any interleaving.

For normal postconditions, the following instructions are grouped in a Bogor atomic block: the return expression (if it exists) is evaluated and the resulting value is assigned to a temporary variable, the postcondition is evaluated (occurrences of `\result` yield the value held in the temporary variable), and the return control action is executed. Without this encoding (e.g., [4]), spurious errors might be reported, for example, if a `put` call is interleaved after a call to `isEmpty` (in Figure 1) returns true, but before the postcondition. For exceptional postconditions, we take advantage of Bogor's built-in exception tables (following the same structure as Java bytecode). In a single atomic block in the exception handler, the exception is caught, assertion is checked, and then the exception is re-thrown. Figure 4 does not show the check of exceptional conditions, but it does show at location `locSpec5`, how the last instruction of the method is aggregated with the checking of the postcondition.

### *4.7 Checking Invariants*

A JML `invariant` is checked in "visible states" as described in Section 2. Note that this means that the notion of invariant in JML is relaxed with respect to the notion of invariant used in model checking and other formal methods where invariants are required to hold in *every* state. Also, in JML at every visible state the invariant for *all* the objects present in the heap are checked; this can be very expensive. We make the observation that given modern programming practice most methods will only modify a very small portion of the program heap. Therefore, a naive enforcement of JML invariants would waste considerable effort in checking invariants on objects that have not changed their state.

Based on this observation, our framework reduces the number of objects on which invariants are checked at visible states, such that only instances of class $C$ are checked at the visible states of methods belonging to $C$. This is problematic only for classes that have public fields that are written directly by methods defined in other classes. Conceptually, we address this problem by treating public field writes as calls to implicit set methods and we extend Bogor to trigger invariant checking at points that correspond to the visible states of those calls.

Figure 4 illustrates this approach using the invariants for the method `refactoredExtract()`. We recall from Figure 1 the assignable clause for this method:

```
//@ assignable head, head.next.value;
```

`head` is a local field, and `value` is a field of the object returned from `head.next`, which is of type `LinkedNode`, which is a class that has no invariants. Hence, in this method we only check the invariants of the class `LinkedQueue`. The corresponding invariant checks can be seen in Figure 4 at locations `locSpec1` and `locSpec5`. Note that invariant checks are inserted at the beginning and at the end of the method. The check for the first three invariants is straightforward: just an assertion with the corresponding boolean expression. The last assertion, however, which corresponds to the invariant:

```
//@ instance invariant \reach(head).has(last);
```

is more interesting because it involves a complex heap manipulation operator (`\reach`). This is a structural consistency property and states that the bottom of the list is always reachable from the head of the list (remember that this is a linked list).

### *4.8 Checking Postconditions Involving `\old` and `\fresh` Operators*

When the construct `\old`($e$) appears in the postcondition of a method $m$, it yields the value of the expression $e$ evaluated in the prestate of $m$. We will discuss the evaluation of this construct in detail; the issues encountered are representative of the interesting challenges that one faces when trying to implement the semantics of a number of JML constructs. In run-time monitoring, one strategy for providing some level of checking for an `\old` construct appearing in a postcondition $p$ of a method $m$ is to: (a) store the value of $e$ in a special local variable $v_e$ when entering $m$, and then, (b) replace `\old`($e$) with $v_e$ in $p$ [4]. When the expression $e$ includes object references, this approach can require storage of all heap data reachable from $e$ in the prestate, and this could well be a substantial portion of the prestate heap.

In a concurrent setting, an additional complication arises since there can be *multiple* prestates associated with a particular poststate of $m$. For example, when a thread $t$ is ready to execute method $m$, but before $t$ enters $m$, one or more actions from other threads may be interleaved — yielding a succession of states where entrance of $t$ into $m$ could occur from any one of these. While achieving coverage of all of these interleavings is problematic for run-time monitoring, a model checker explores all interleavings of threads, thus, it can naturally check a postcondition with respect to all associated prestates.

Since an explicit-state model checker such as Bogor stores all states, one might think that when arriving at an instance of `\old` in a postcondition, we can simply retrieve the appropriate prestate from the model checker's depth-first-stack of visited states to evaluate the `\old` expression. However, we explain in detail below that it is necessary to store additional information of the method's prestate during its execution to ensure that postconditions are evaluated for all appropriate states. Without storing additional information about the prestate to appropriately distinguish method execution states with different values of `\old`, some contexts in which the postcondition must be checked might be missed because the model checker may end up hitting a state stored in the model checker's seen-before-set of states (and thus backtrack) before it reaches the points of a method where postconditions are checked (and this may happen even though the prestates are different).

Figure 5 presents an example to illustrate this issue. The figure gives a fragment of Java with a simple postcondition and the state-space constructed by Bogor using a depth-first search state exploration with two instances of the `Race` thread. For simplicity, the state is illustrated by a state vector containing four integers: (1) the value of the static variable `x`, (2) the PC of the main thread, (3) the PC of the first instance of `Race`, and (4) the PC of the second instance of `Race`. We denote the PC of threads at loc$X$ as the integer $X$. The first

```
class RaceDriver {
  public static final main(String[] args) {
    Race r1 = new Race();
    Race r2 = new Race();
    r1.start();
    r2.start();
  }
}

class Race extends Thread {
 private static int x;

 public void run() {
  loc1: x = 0;
  loc2: foo();
 }

 /*@ ensures
   @  \old(x) == 0;
   @*/
 private void foo() {
  loc3: x = 1;
  loc4: return;
 }
}
```

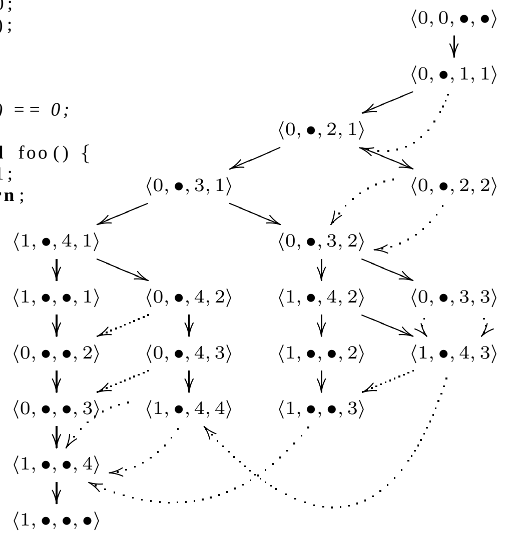

**Fig. 5.** Race Example and its DFS state-space

location of the main thread is `loc0`. We use • to denote the PC of a thread that has died or has not been created yet. For transitions between state vectors, a straight arrow denotes an atomic step in the model checker, and a dotted arrow denotes an atomic step that causes the model checker to backtrack because it has seen the destination state (i.e., the destination state is stored in the model checker's seen-before-set).

To reduce the state-space of the example, we use the thread symmetry and partial-order reduction techniques described in Section 3. Thread symmetry causes, for example, the state $\langle 0, \bullet, 1, 2\rangle$ to be considered equivalent to the state $\langle 0, \bullet, 2, 1\rangle$. Partial-order reduction causes all the transitions of the main thread to execute without any interleaving of the newly created `Race` instances. Note that the reductions do not affect the result of checking the postcondition; the problem that we are presenting also occurs in the unreduced state-space.

In our implementation strategy for checking postconditions, the execution of a `return` statement is aggregated with the transition that checks the postcondition so that the two transitions are executed in a single atomic transition. Thus in the example of Figure 5, the postcondition `\old(x) == 0` is checked immediately following the execution of the `return` statement at `loc4`.

The following trace through the state-space of Figure 5 violates the postcondition:

$$\langle 0,0,\bullet,\bullet\rangle \rightarrow \langle 0,\bullet,1,1\rangle \rightarrow \langle 0,\bullet,2,1\rangle \rightarrow \langle 0,\bullet,3,1\rangle \rightarrow \langle 0,\bullet,3,2\rangle \rightarrow$$
$$\langle 0,\bullet,3,3\rangle \dashrightarrow \langle 1,\bullet,4,3\rangle \dashrightarrow \langle 1,\bullet,\bullet,3\rangle \dashrightarrow \langle 1,\bullet,\bullet,4\rangle \dashrightarrow \langle 1,\bullet,\bullet,\bullet\rangle$$

Specifically, at step $\langle 1,\bullet,\bullet,4\rangle \dashrightarrow \langle 1,\bullet,\bullet,\bullet\rangle$ the postcondition will fail to verify because the value of `x` at one of the prestates of the second instance of `Race` (i.e., $\langle 1,\bullet,\bullet,3\rangle$) is non-zero. However, this violating trace is not found by the state space exploration because the atomic step $\langle 0,\bullet,3,3\rangle \dashrightarrow \langle 1,\bullet,4,3\rangle$ causes Bogor to backtrack as it has already seen the state $\langle 1,\bullet,4,3\rangle$ from a different trace. Thus, the subsequent steps (including the postcondition check) in the error trace are not encountered in the state-space exploration.

The problem here is reminiscent of issues associated with temporal logic model checking in which state information associated with the property being checked needs to be added to system state information to ensure complete checking with respect to the property being considered [5, p. 122]. We solve the problem in a similar way by identifying a portion of the prestate that can be used to distinguish identical poststates that are arrived at from different prestates; it suffices to consider the set of objects reachable from references that are visible in the prestate. This calculation can be performed efficiently in Bogor because: (1) as discussed Section 3, Bogor employs collapse compression techniques that reuse representations of objects from previous states when storing a new state, and (2) we can augment the thread state that will execute the postconditions containing `\old`($e$) by a collapsed state fragment that encodes the relevant prestate objects. Conceptually, our approach is analogous to adding the collapsed prestate fragment as a local variable of the method, for example,

```
private void foo() {
  int collapsedState = Bogor.getCollapsedState(e);
  loc3: x = 1;
  loc4: return;
}
```

where method `Bogor.getCollapsedState(e)` returns the unique collapsed state id of the object referred to by $e$ (and all objects reachable from $e$). Intuitively, this makes identical poststates that derive from different prestates (with respect to $e$) distinguishable from each other. In general, this addition to the state space might cause significant increase in checking time and space, but as our experimental evaluations in Section 5 demonstrate, this can be mitigated through the use of reduction techniques that detect and exploit atomic method execution as determined by partial order reductions [14]. These reductions prevent interleaving in sequences of bytecodes that are independent of bytecodes from other threads – causing the sequence of bytecodes to execute in a single atomic step. The lack of interleaving enables our POR framework to employ an additional optimization: instead of storing the intermediate states when executing an atomic bytecode sequence, we only store the final state produced at the end of the atomic sequence. For methods whose bodies consist entirely of independent actions, the atomic block runs from the precondition through the postcondition. This means that the fingerprints of the relevant prestate that would normally be recorded are never stored in a state since they are consumed in the postcondition before a state is stored at the end of the atomic block.

Now let us describe how the concept above is actually implemented at the BIR level. The post condition of `refactoredExtract()` from Figure 1 is:

```
//@  ensures \result == null || (\exists LinkedNode n;
//@          \old(\reach(head)).has(n);
//@                n.value == \result
//@                && !(\reach(head).has(n)));
```

The second condition in the disjunct makes a reference to the prestate value of all the objects reachable from `head`. Thus, we need to store, at the beginning of the method, the collapsed portion of the heap that is relevant for this property — in this case, the portion of the heap reachable from `head`. For this, we extend BIR with a function that, given a reference, returns the collapsed encoding of the heap reachable from the reference argument. Figure 4 illustrates the use of this extension function in the first instruction of location `locSpec1` to store the collapsed heap ID in the local variable `collapsedState`.

We now turn to the issue of how to evaluate the arguments of `\old` constructs when evaluating postconditions. We have described above how we save the *collapsed* (i.e., optimized, compacted) representation of `\old` argument values in the state vector being used to evaluate a method. One might imagine using those saved collapsed values to directly evaluate the arguments of `\old` constructs. We avoided this approach for two reasons. First, the Bogor API for the state manager component does not provide a method mapping from a compressed state to an uncompressed state *by default* (i.e., it is not mandated by the API) to allow flexibility of usage of various kinds of loss-less and lossy compression (e.g., hashing using MD5) algorithms. Second, and most importantly, uncompressing states that have been generated by a sophisticated compression algorithm, such as our implementation of collapse compression, can be very expensive. Thus, uncompressing states is avoided so as to not further increase the cost of model checking.

Therefore, we take a different approach for retrieving the values from the prestate needed to evaluate the arguments of `\old`. In essence, when evaluating the postcondition, we use the existing model checker infrastructure for undo-ing state transformations to back up to a prestate, and the `\old` arguments are evaluated in that state. We abstract the steps required for this by introducing another extension function called `State.preVal` whose use is illustrated in the implementation of postcondition evaluation in Figure 4. Given the representation of an expression to evaluate, `State.preVal` calculates the prestate value by using the backtracking facilities of Bogor: starting at the poststate, it clones the state and backtracks to the prestate, evaluates the expression, and returns the result of the evaluation (and throws away the cloned and the backtracked state). This result value is stored in a temporary variable (`spec1` in Figure 4) and then used subsequently in the postcondition checking.

An alternate implementation strategy would be to propagate *forward* from method entry the required uncompressed prestate values for evaluating `\old` in postconditions. However, a preliminary investigation of this strategy indicated that it would be more expensive because not all traces beginning from the method entry will be fully explored up to the method exit where the postcondition is checked; Bogor may backtrack earlier because it has seen the middle states. By employing the backing up strategy described above, we ensure the prestate is queried only when it is actually needed. In addition, in a multi-threaded program, Bogor may check several methods at the same time. This means that in some cases memory consumption is larger due to the forward propagation of several prestates. In the backward approach, this is always done one method at a time, on-demand basis.

JML's `\fresh`$(x_1, \ldots, x_n)$ operation requires that, at the poststates of a method $m$, variables $x_i$ are non-null and the objects bound to $x_i$ are not present in any of the prestates of $m$. For `\fresh`, we explicitly store newly allocated object references in a method local variable to minimize the stored state information, since the number of allocated objects in a method activation is usually significantly smaller than the set of all objects in the prestate. To handle method call chains (a method calls a method that allocates an object to a reference declared fresh by the first method), we use an approach similar to the one used to check frame conditions that is described in the next sub-section: a method that declares some fresh variable, instructs the model checker to create a set (the fresh set) to keep track of newly allocated objects, and which is passed down the call chain.

### *4.9 Checking Frame Conditions: The `assignable` Clause*

The `assignable` $ap_1, \ldots, ap_n$ annotation for a method $m$ specifies that the field/variable given by the access path $ap_i$ can be assigned during the execution of $m$. According to the JML definition, each access path $ap_i$ must have the form $x.f_1 \ldots f_k$, where $f_i$ is either a field or an array access, and where $k > 0$; access paths with null-prefixes are ignored.

The assignable clause is difficult to check without being overly conservative due to the presence of aliasing, and consequently many tools simply avoid this check. Since Bogor is an explicit-state model checker, its explicit representation of the heap has complete alias information. Thus, it can decide precisely whether an assignment satisfies the assignable clause. When an assignable clause is specified for an access path $x.f_1 \ldots f_k$, we extend Bogor so that it records that the field $f_k$ of the object represented by $x.f_1 \ldots f_{k-1}$ (when entering $m$) may be assigned during the execution of $m$; any assignment to the heap in the body of the method that has not been thusly recorded is flagged as an error.

In addition, for nested method calls, the semantics of the `assignable` clause requires that the sets of assignable locations of a nested method are a subset of those for any enclosing method. Again, Bogor can easily check this on-the-fly since its explicit heap representation keeps precise alias information.

The mechanism for checking frame conditions with Bogor is illustrated in Figure 6. To control and monitor the assignments to any portions of the heap we needed to modify the behavior of the module that takes care of performing these assignments. This module is the `IActionTaker` (shown in Figure 2), which interprets BIR actions (i.e., commands). We modified this module to implement the stack-like mechanism shown in the figure.

Basically, the information about which variables can be assigned to, at any given moment, are maintained in a per thread stack data structure. Each level of the stack corresponds to the frame conditions of a specific method and the depth of the stack is the depth of the method calls chain. Every time an assignment is made, there is a check to verify that the memory location is in the stack. Only the top level is inspected because Bogor enforces the following correctness condition on the stack: the set of memory locations at any level of the stack (the memory locations that can be assigned by the method associated with the frame) must be a subset of the set of memory locations of the stack level immediately below, with the exception of the bottom level which has no restrictions. If this condition is violated for any method, an error message is re-

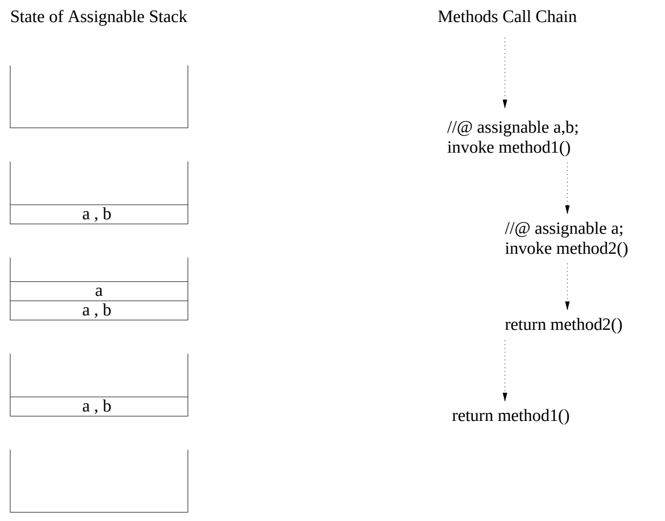

**Fig. 6.** Mechanism used to check frame conditions in Bogor.

ported. This enforces the JML restriction that a method $m_1$ cannot call another method $m_2$ that assigns to locations not listed in $m_1$'s assignable clause.

This mechanism is implemented in BIR using a set of extension operators. We can see in Figure 4 how the verification of frame conditions is translated to BIR. First, in location `locSpec4` we see the following instructions:

```
State.enterAssignable();
State.addAssignable<(|linkedqueue.LinkedNode|)>(...);
```

The first instruction is executed to inform to the extended `IActionTaker` that a new stack level must be allocated. The second instruction is used to add objects to the assignable list of this method. These instructions should be executed at the beginning of the method, before the body of the method starts executing. Finally, when the method finishes, we let the `IActionTaker` know that this method has finished, so the frame conditions for this method can be popped from the stack, as seen in location `locSpec5`:

```
State.exitAssignable();
```

At this point all the information corresponding to the frame conditions for the method are eliminated and what is left is the assignable memory locations from methods up in the method call chain. This instruction is inserted, of course, after all the method body instructions have been executed.

### *4.10 Methods in JML Expressions*

It is convenient to allow JML expressions to invoke Java methods as “helper expressions”. Semantically, this is only sound if the method does not change the observable state. The method annotation `pure` declares that a method $m$ is side-effect free. The JML definition of *pure* is that $m$ does not diverge and its assignable clause is empty. Note that this definition does not allow methods annotated as pure to synchronize on objects as this would represent a modification of an object's lock state. For example, the `isEmpty` method in Figure 1 cannot be declared as a pure method. However, it would be useful to consider methods such as `isEmpty` as pure, as discussed in [21].

To address this, we introduce the notion of *weak purity*. The intuition is that a weakly pure method can contain assignments (e.g., to local variables or newly allocated objects that do not escape the method) as long as the state observed by other threads does not change. In other words, in a context in which the method is executed without any interleaving, the poststate of the method should be identical to the prestate (modulo differences in the PC for the executing thread).

Using this definition, the `isEmpty` method can be considered as weakly pure. This condition can be checked in Bogor by comparing the prestates and poststates of methods as they are called. We implement and use this kind of weak purity in our framework in order to be able to call methods such as `isEmpty` from within JML expressions. This can be done because of the non-interfering property between the verification of the specification of a system and the execution of the system itself, described in Section 4.2. That is, even if `isEmpty` has temporary side effects (i.e., the lock is acquired), these effects are invisible to the system as long as the initial state is restored after `isEmpty` has finished execution, because no other thread is interleaved with the execution of specification checks.

## 5 Evaluation

This section overviews the methodology and important issues that need to be considered in practical application of our framework and then provides a summary of the performance of the framework on a collection of Java programs.

### *5.1 Tool Methodology*

The exhaustive exploration of a system's state space inherent in model checking makes it difficult to scale to large systems. Many researchers believe that it is most natural to apply model checking to software units, or modules, instead of whole systems. Specifically, software model checking is often envisioned as part of a development and quality assurance methodology in which it is incorporated with *unit testing*. When model checking a software unit, one typically desires to specify/check as much of the unit's behavior as possible in the hope of detecting as many bugs as possible.

To apply model checking to software units (with or without JML specifications), a developer needs to follow an approach that is similar in many respects to the steps involved in traditional unit testing. In unit testing, one develops a *test harness* that makes method calls into the unit for specific sets of parameter values and examines the results of the method calls for invalid results (indicating failed test). When applying model checking to software modules, one must similarly use a test harness (also termed a *closing environment* or an *environment*) to drive the unit through particular execution paths. The scale and complexity of a software unit's interface may vary greatly: a unit may consist of multiple classes and interfaces that expose fields and methods through a variety of mechanisms, such as, reference, method call, inheritance and interface implementation. Consequently, a general test harness for a unit must be designed to accommodate all legal modes of external interaction. The environment will sequence those interactions to represent the behavior of program contexts in which the unit will be used in a larger piece of software.

It is important to note that for theorem-proving-based JML checking tools like ESC/Java and LOOP, verification of a method's behavior against a JML specification proceeds, at a conceptual level, by assuming the precondition and showing that the postcondition of a method is satisfied. This ensures that method is verified against the JML specification for any context in which the method could ever be called. Thus, method preconditions (along with class invariants) effectively describe the set of program states against which method bodies will be verified, and when the tool reports that a software unit is "verified", the user can conclude that a method's postcondition will be satisfied for all method prestates that satisfy the method's precondition.

In contrast, with a model checking or run-time monitoring approach to JML verification, JML specifications for a method are not checked against all possible states that satisfy the precondition nor against all possible invocation contexts, but only for the invocation contexts and method prestates generated by the given test harness that happen to satisfy method preconditions. Thus, when the tool reports that a software unit is "verified", the user cannot conclude that a method's postcondition will be satisfied for all method prestates that satisfy the method's precondition, but only for the states generated by the given test harness and filtered by method preconditions. However, a key difference between run-time monitoring and model checking in this context is that model checking will explore all concurrent interleavings generated by the test harness whereas run-time monitoring will only explore one trace for each verification run.

Various abstraction techniques such as counterexample guided predicate abstraction refinement are often used in software model checking [1,15]. While those techniques have been applied with good success for verifying simple temporal safety properties in the context of sequential C programs with relatively little heap-allocated data, they have yet to be effectively applied in the context of concurrent software with large amounts of heap data and rich behavioral specifications such as those one typically encounters in JML-annotated Java programs. Thus, our approach to obtaining tractable state spaces focuses on *bounding* the program's behavior in various ways. This includes limiting the number of threads and size of data structures generated in test harnesses, selectively introducing representative data elements for contents of data structures, and bounding the ranges of integer variables. Of course, this type of bounding also limits the range of behaviors for which the software unit is checked. Thus, there is a delicate balance in constructing test harnesses and model bounds so that a wide range of system behaviors are checked while constructing the space, enough to yield tractable checking.

In conclusion, our JML checking approach targets developers who are interested in automated methods for finding bugs by checking rich JML specifications against program modules written in full featured multi-threaded Java where the modules being checked are of the size typically considered in unit testing. Section 6 provides a broader discussion of the trade-offs between existing approaches to checking JML specifications.

## *5.2 Experiments*

We applied Bogor to reason about six Java programs, most of which are multi-threaded and manipulate non-trivial heap-allocated data structures. Table 2 reports several measures of program size: **loc** is the number of control points in the source text, **threads** is the number of threads of control in the instance of the program, and **objects** is the maximum number of allocated objects on any program execution. All programs were annotated with JML invariants, pre/postconditions, and assignable clauses; the table highlights the challenging features used in the specifications for each program. We report the number of states visited during model checking as well as machine dependent measures, the run-time in seconds and memory in mega-bytes of RAM, for each version of

| Program | POR and JML | POR and no JML | no POR and JML | no JML and no POR |
|---|---|---|---|---|
| BoundedBuffer[13] | 164 loc | `\fresh`, `\old`, `signals` | | |
| 3 threads | 69 states | 69 states | 2647 states | 2647 states |
| 10 objects | 1 sec/0.8 MB | 1 sec/0.6 MB | 4 sec/1.2 MB | 3 sec/1.0 MB |
| 7 threads | 1098 states | 1098 states | 1601745 states | 1601745 states |
| 18 objects | 26 sec/1.2 MB | 23 sec/1.0 MB | 8936 sec/180.2 MB | 8458 sec/167.7 MB |
| DiningPhilosopers[13] | 193 loc | `\forall`, `\fresh` | | |
| 4 threads | 38 states | 38 states | 12514 states | 12514 states |
| 6 objects | 1 sec/1.1 MB | 1 sec/0.7 MB | 27 sec/2.5 MB | 20 sec/2.0 MB |
| 6 threads | 1712 states | 1712 states | 1939794 states | 1939794 states |
| 8 objects | 32 sec/2.3 MB | 24 sec/1.8 MB | 9571 sec/159.9 MB | 8719 sec/157.6 MB |
| LinkedQueue[19] | 228 loc | `\fresh`, `\reach`, `\old`, `signals`, `\exists` | | |
| 3 threads | 2833 states | 1533 states | 17064 states | 11594 states |
| 22 objects | 10 sec/1.6 MB | 5 sec/1.0 MB | 38 sec/3.7 MB | 21 sec/2.3 MB |
| 5 threads | 39050 states | 12807 states | 1364007 states | 423538 states |
| 32 objects | 144 sec/5.9 MB | 72 sec/2.5 MB | 14557 sec/140.5 MB | 2415 sec/46.4 MB |
| RWVSN[19] | 227 loc | `\old` | | |
| 4 threads | 183 states | 183 states | 2621 states | 2255 states |
| 5 objects | 1 sec/1.0 MB | 1 sec/0.8 MB | 2 sec/1.5 MB | 2 sec/1.0 MB |
| 7 threads | 18398 states | 18398 states | 4995560 states | 4204332 states |
| 9 objects | 185 sec/6.8 MB | 144 sec/3.0 MB | 34804 sec/463.7 MB | 26153 sec/366.3 MB |
| ReplicatedWorkers[8] | 543 loc | `\fresh`, `\old`, `\reach` | | |
| 4 threads | 1751 states | 1751 states | 322016 states | 269593 states |
| 19 objects | 14 sec/2.1 MB | 13 sec/1.9 MB | 897 sec/29.8 MB | 716 sec/26.6 MB |
| 6 threads | 10154 states | 10154 states | 12347415 states | 10016554 states |
| 21 objects | 99 sec/3.3 MB | 92 sec/2.8 MB | 30191 sec/391.8 MB | 21734 sec/282.5 MB |
| java.util.Arrays.sort(Object[]) | 151 loc | `\forall`, `\exists`, `\old` | | |
| 1 thread | 2 states | 2 states | 21597 states | 21597 states |
| 502 objects | 82 sec/2.0 MB | 7 sec/1.9 MB | 391 sec/49.5 MB | 343 sec/48.8 MB |

**Table 2.** Checking time/space for JML Annotated Java Programs

the example programs; data was gathered running Bogor under JDK 1.4.1 (32-bit mode) on a 2 GHz Opteron with maximum heap of 1 GB running Linux (64-bit mode).

In the following subsections, we give a brief description of each of the six programs used in the experiments and the driver used to perform each experiment. As mentioned before, Table 2 gives details about further configuration of the test drivers: number of threads and number of objects. We also give an overview of the kind of properties verified in each system. Source code and BIR models for all the experiments can be found in [33].

### 5.3 *BoundedBuffer*

This program is an implementation of a concurrent buffer, using a fixed size array, obtained from [13]. The program has four classes: `BoundedBuffer`, `Consumer`, `Producer`, and `ProducerConsumer`.

The main class is `BoundedBuffer`. This class has a constructor that initializes the underlying array to the initial bound (the number of slots). The class declares two methods: `deposit(Object)` and `fetch()` which are used to insert and extract objects to/from the array, respectively. These methods are synchronized and implement a blocking policy.

The `Consumer` and `Producer` classes just implement threads that fetch and deposit objects from/to the buffer, respectively. Finally, the `ProducerConsumer` class just initializes several threads of each type and starts the run.

The specifications in this program are mostly located in `BoundedBuffer` and focus on checking frame conditions (`assignable`) and structural invariants, such as checking that the size of the buffer always keeps between bounds.

### 5.4 *DiningPhilosophers*

This is an implementation of the dining philosophers problem obtained from [13], with just three classes: `DiningServer`, `Philosopher`, and `DiningPhilosophers`.

The `DiningServer` class can be thought of as providing the *dining table*: it provides the abstraction of the number of forks and keeps track of their state (held or free). It also provides functions for picking up and releasing the forks. The `Philosopher` class implements a thread that interacts with the server, always trying to pick up the forks, and going into a *thinking* state if it cannot do so. `DiningPhilosophers` is the driver class: it initializes the philosophers and starts the run.

The specifications in this program have to do mostly with correctness invariants, for example, the philosophers cannot be eating all at the same time, two adjacent philosophers cannot be eating at the same time, and in order to pick up a fork it must be free.

### *5.5 LinkedQueue*

This program has already been described in Section 2, so we only briefly describe some other details in this section. There are three classes:`LinkedQueue`, `LinkedNode`, and `LinkedQueueDriver`.

The `LinkedQueue` class has already been explained with detail in Section 2. The class `LinkedNode` provides the node structure for the linked list used by the queue. Finally, the `LinkedQueueDriver` is the driver class: it initializes a group of threads that insert and remove elements to/from the queue, and starts the run.

### *5.6 RWVSN*

This program is an implementation of a readers-writers lock obtained from [19]. The program has four important classes: `RWVSN`, `Reader`, `Writer`, and `RWVSNDriver`.

The `RWVSN` class implements the readers-writers lock. `Reader` and `Writer` implement threads that try to read and write to the resource protected by the readers-writers lock, respectively. `RWVSNDriver` is the driver class that starts a group of reader and writer threads, on the same resource, and starts the run.

The specifications for this program include the normal global invariants for a readers-writers lock, for example, at all times there is at most one writer accessing the resource, and if there is a writer accessing the resource no readers can be accessing it.

### *5.7 ReplicatedWorkers*

This program provides an abstraction to implement a set of threads that perform a given task according to a given policy that all the threads follow (hence, replicated worker threads). This program was obtained from [8]. The program has a total of 16 classes, of which we only list the most important ones: `ReplicatedWorkers`, `Coordinator`, `Worker`, and `BasicRWTest`.

The `ReplicatedWorkers` class is a factory for creating, initializing, executing and terminating a parallel computation that is factored into a collection of identical sub-computations operating on partitions of data. The class `Worker` encapsulates the sub-computation to be performed on a data partition. The `Coordinator` class controls the dispatch of data partitions to worker threads and the accumulation of results returned from those threads. Finally the `BasicRWTest` class is a driver class that initializes a set of workers with a set of initial tasks, and starts the run. For more details we refer the reader to [8].

The specifications in this program are focused on synchronization behavior and global invariants based on the synchronization policy.

### *5.8 java.util.Arrays.sort(Object[])*

We wanted to try the technique on a sorting algorithm, and we picked Java's array sort function which is a sequential method. Even so, it shows the kind of powerful specifications that can be written in JML.

This test has only one class: `ComparableSort`. This class is a driver that allocates an array with elements unsorted, and then calls the `sort` method on the array.

The specifications for this method include structural consistency properties on the array, such as, the element count is the same at the beginning and at the end, and the elements in the array are the same elements that the array had before the sort. There are also specifications of the sorting property: at the end, the elements are sorted in non decreasing order.

### *5.9 Discussion*

For each program version, we ran model checks for each of the four combinations of object-sharing based partial order reductions (POR) and JML checking features. By comparing runs with and without JML checking, one can determine the overhead of JML checking. For half of the examples, regardless of the use of reductions, the use of potentially problematic features like `\old` and `\fresh` yields no significant overhead for JML checking. Without POR, however, there is non-trivial overhead for three of the six programs; in the worst-case, LinkedQueue, space consumption increased by a factor of three and time by a factor of six. This is not unexpected since the JML specifications for LinkedQueue contain

`\reach` expressions within a `\old`; consequently nearly all of the prestate heap must be used to distinguish poststates. Comparing runs with and without POR reveals the significant benefit of sophisticated POR; it yields between 2 and 4 order of magnitude reductions in the size of the state space on our set of example programs. Furthermore, the use of POR significantly mitigates JML checking overhead. For the worst-case example, run-time overhead is reduced from a factor of six to a factor of two. For the RWVSN and ReplicatedWorkers, the fact that these programs have methods with atomic execution behavior allows our POR to eliminate nearly all of the JML checking overhead. Only when methods are not atomic does JML checking suffer significant overhead.

The increase in complexity of JML brings an increase in the possibility of making errors in writing specifications. In the presence of concurrency, it is not uncommon to make subtle errors in defining the intended behavior of a class or method. We experienced this in annotating several of the examples used in our study. As has been observed by others, we found that the generation of counterexamples proved to be extremely useful in debugging erroneous specifications. The exhaustive nature of the search in model checking makes it a more effective *specification debugging* tool than run-time checking.

We included a standard comparison sorting program in our set of examples to illustrate Bogor's behavior on a declarative JML specification of a rich behavioral property (i.e., the poststate is an ordered permutation of the prestate). Despite the richness of this specification, because of the singly threaded nature of the program the method trivially executes atomically, thus, there is no overhead for JML checking. Our on-the-fly atomic block detection algorithm dramatically reduces the number of states, memory and time required for analysis since it defers the storage of a global state until the *current* thread reaches a point where it modifies data that can be observed by another thread. Since there are no other threads, this only happens at the final program state, hence the second state.

## 6 Related Work

Burdy et. al. [3] survey the steadily growing body of tool support for reasoning about JML specifications. In general, there are three underlying technologies used in these tools: semi-automated theorem proving, automated decision procedures, and run-time monitoring. These technologies have different advantages and disadvantages which we assess along four dimensions:

**Automation/Usability** How much effort is needed to use the technology or tool?

**JML Coverage** How much of the JML language is supported?

**Behavior Coverage** How much of a program's behavior is considered in reasoning?

**Scalability** How does reasoning cost grow with system size and complexity?

In this section, we characterize the strengths and weaknesses of the basic technologies in terms of these dimensions. While we cite specific tools that implement JML reasoning with those technologies, the strengths and weaknesses mentioned are, for the most part, characteristics of the underlying technology. We note that despite their weaknesses, each of these techniques and associated tools is *useful* in that they have been used effectively on real Java programs.

LOOP [38] is the most mature theorem-prover-based JML reasoning system. It translates JML specifications and Java source code into proof obligations for the theorem prover PVS [24]. Thus, the semantics of the Java code as well as JML specifications are represented as PVS theories, and users verify specifications against the Java code by interacting with the PVS command-line interface to discharge the generated proof obligations. LOOP is difficult for novices to use since it requires detailed knowledge of logical representation of Java semantics. Recent advances in LOOP's weakest-precondition calculus allow methods with straight-line code performing integer calculations to be verified with little or no user intervention by leveraging the underlying numerical procedures of PVS. LOOP scales poorly to general Java applications due to the complexities of its logical treatment of aliasing. With sufficient expertise, however, LOOP allows very strong correctness properties to be established with the highest-possible degree of confidence.

ESC/Java [10] is another decision-procedure-based tool for a subset of JML. ESC/Java allows the user to work at the Java level by encapsulating the translation of verification conditions to an underlying decision procedure. It gains a high degree of automation by treating a small subset of JML and by sacrificing soundness in the results of its analysis. ESC/-Java targets the efficient, automatic checking of null references and simple integer properties (e.g., array bounds violations), but does not support richer properties, for example, those that require quantification over class instances or any of JML's heap related primitives. It uses a modular checking approach in which methods are verified in isolation by trying to prove that class invariants and method postconditions hold under the assumption that the method's preconditions are satisfied. ESC/Java is fully automatic and its modular checking approach allows it to scale to large programs (e.g., up to 50K lines of code) in terms of run-time, but experience suggests that significant user effort is required to annotate a program sufficiently to enable verification of properties. A new version of ESC/Java that supports more of JML is available at [23].

Cheon and Leavens [4] have developed a run-time checker (`jmlc`) which compiles JML-annotated programs into bytecode that includes run-time assertions to check JML specifications. As with other run-time analysis methods, reasoning using `jmlc` requires a complete Java program, thus if a single class or method is to be analyzed, an appropriate test harness must be constructed. Aside from this, using `jmlc` is fully automatic for a good portion of the JML language; notably lacking are general support for class instance domains and access to precondition state values in postconditions which are very common in heavyweight behavioral specifications. `jmlc` implements run-time checking on top of existing JVMs and consequently it provides no direct support for multi-threaded programs.

We believe that our approach to model checking JML specifications using a model checker that is customized to exploit JVM semantics has a different set of strengths and weaknesses than these techniques. It is as automated as run-time checking, but provides significantly greater JML and behavior coverage. It provides a high-degree of JML coverage as do theorem-prover approaches, but it does not suffer as seriously from usability problems, in terms of providing user guidance. Since model checking is a whole-program analysis, it does not suffer the annotation-burden of modular decision-procedure approaches. While model checking is not as scalable as run-time analysis or modular decision-procedure based approaches, it is capable of treating thousand-line Java programs with embedded JML specifications which is sufficient for non-trivial unit-level or subsystem-level program verification.

### *6.1 Other Related Work on Specification Languages*

There have been an enormous number of efforts to define languages for specifying and reasoning about program behavior. We are interested in providing automated reasoning support for strong properties of modern concurrent object-oriented languages, thus, discussion on most of the existing work on simple assertion languages and manual formal methods is lacking. Recent work on OCL and Alloy is aimed at supporting at least some of our goals.

Space constraints do not permit a detailed discussion of the different checking mechanisms that have been proposed for OCL. One line of work, e.g., [17], is similar to `jmlc` in that it generates run-time assertions for checking Java. Another popular direction is to compare OCL specifications with other UML models, e.g., [27], rather than program source code, thus a number of the issues regarding reasoning about heap-allocated program data are not considered in that work.

The Alloy Annotation Language (AAL) [18] is a language for annotating Java code with a syntax that is similar to JML. AAL supports analysis, via bounded satisfiability checking, of loop-free code sequences that may have method invocations. AAL targets the verification of small methods that maintain invariants on complex heap structures (e.g., red-black trees). The Java heap is modeled in Alloy using relations, and checking is carried out automatically by generating all possible heap-structures that can be constructed from a user-bounded set of objects. AAL does not support reasoning about concurrent programs.

## 7 Conclusion

For model checking to become useful as a software validation technique it must become *more efficient* (so that it provides feedback on fragments of real code in a matter of minutes), *more expressive* (so that it can reason about a broad range of functional properties of interest to software developers), and *more standardized* (so that developer investment in writing specifications can be leveraged for multiple forms of validation and documentation). In this paper, we have presented an approach to customizing a model checking framework to meet these goals by incorporating novel state-space reductions and support for reasoning about behavioral properties written in JML. Our data suggest that the combination of these techniques can provide cost-effective reasoning about non-trivial properties of multi-threaded Java programs.

Bogor supports nearly all of JML, but there are a few features that we are still working to support. Chief among these are JML's *model programs* which we are implementing as Bogor type extensions that directly encode the model abstract data types (ADTs) as first-class types in the Bogor's input language. We believe this alternative will be more efficient than [4] because we can apply various reduction algorithms to those ADTs, as shown in [28].

There are several issues that need to be addressed to increase the effectiveness of this approach. In Section 5, we discussed the need to provide test harnesses for model checking software units. The problem of constructing such test harnesses is often overlooked in the research community and is surprisingly difficult. For example, Penix et al. [25] note that it took several months to construct an environment that correctly modeled the context of the DEOS real-time operating system scheduler. In addition, in a proper methodology a user should seek to evaluate the completeness of test harnesses for covering all the behaviors referred to by the specification, since (as described above) the software unit is only checked up to the behaviors generated by the test harness.

Driven in part by the work presented in this paper, we have developed multiple forms of tool support to address these issues. First, the Bandera Environment Generation (BEG) tools [37,36] provide basic support that addresses many of the challenges encountered in the DEOS case study, but it does not treat the full range of complexities that arise when model checking units with JML specifications. To assess the quality of test harnesses with respect to JML specifications, we have built a coverage framework [31,34] on top of Bogor to provide information about the portions of code and JML specifications that are actually exercised by a given test harness. An important consideration in this process is the selection of input values to achieve thorough coverage of the behavior of both the program and the specification, and we are extending recent work by Stoller [35] towards that end.

Finally, we believe that a natural direction for future work on model checking JML specifications is to enrich JML to support the explicit specification of concurrency related behavior. One proposal for JML concurrency extensions is centered around introducing the notion of atomicity in method behavior specifications [30]. Our initial experience adapting existing approaches to checking atomicity properties via model checking [14,9] suggests that reasoning about JML specifications that explicitly capture concurrency properties along with behavior properties of programs is feasible.

## References

1. T. Ball and S. K. Rajamani. The slam toolkit. In *Proceedings of the 13th International Conference on Computer Aided Verification*, pages 260–264. Springer-Verlag, 2001.
2. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.
3. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In *Proceedings of the Eighth International Workshop on Formal Methods for Industrial Critical Systems*, volume 80 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.
4. Y. Cheon and G. T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *Proceedings of The International Conference on Software Engineering Research and Practice*, pages 322–328. CSREA Press, June 2002.
5. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Jan. 2000.
6. M. B. Dwyer, J. Hatcliff, Robby, and V. R.Prasad. Exploiting object escape and locking information in partial order reduction for concurrent object-oriented programs. *Formal Methods in System Design*, 25(2–3):199–240, September–November 2004.
7. M. B. Dwyer, Robby, X. Deng, and J. Hatcliff. Space reductions for model checking quasi-cyclic systems. In *Proceedings of the 3rd International Conference on Embedded Software (EMSOFT 2003)*, volume 2855 of *Lecture Notes in Computer Science*, pages 173–189. Springer, October 2003.
8. M. B. Dwyer and V. Wallentine. A framework for parallel adaptive grid simulations. *Concurrency : Practice and Experience*, 9(11):1293–1310, November 1997.
9. C. Flanagan. 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*, pages 252–266. Springer, April 2004.
10. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM Press, 2002.
11. R. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Pub. Co., Jan. 1995.
13. S. Hartley. *Concurrent Programming - The Java Programming Language*. Oxford University Press, 1998.
14. J. Hatcliff, Robby, and M. Dwyer. 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*, pages 175–190. Springer, 2004.
15. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 58–70, Jan. 2002.
16. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
17. H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In *The Third International Conference on The Unified Modeling Language*, volume 1939 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2000.
18. S. Khurshid, D. Marinov, and D. Jackson. An analyzable annotation language. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and ap-*

*plications*, pages 231–245, New York, NY, USA, 2002. ACM Press.
19. D. Lea. *Concurrent Programming in Java: Second Edition*. Addison-Wesley, 2000.
20. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Speciﬁ cations of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.
21. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. Cok. How the design of JML accommodates both runtime assertion checking and formal veriﬁ cation. In *Proceedings of the 1st International Symposium on Formal Methods for Components and Objects*, volume 2852 of *Lecture Notes in Computer Science*, pages 262–284. Springer, November 2002.
22. B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.
23. NIII. ESC/Java2 Website. `http://www.cs.kun.nl/sos/research/escjava/index.html`.
24. S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype veriﬁ cation system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artiﬁ cial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.
25. J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Veriﬁ cation of time partitioning in the deos scheduler kernel. In *Proceedings of the 22nd international conference on Software engineering*, pages 488–497. ACM Press, 2000.
26. A. D. Raghavan and G. T. Leavens. Desugaring JML method speciﬁ cations. Technical Report 00-03d, Iowa State University, Department of Computer Science, July 2003.
27. M. Richters and M. Gogolla. Validating UML models and OCL constraints. In *The Third International Conference on The Uniﬁ ed Modeling Language*, volume 1939 of *Lecture Notes in Computer Science*, pages 265–277. Springer, 2000.
28. Robby, M. B. Dwyer, and J. Hatcliff. 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*, pages 267–276. ACM Press, 2003.
29. Robby, M. B. Dwyer, J. Hatcliff, and R. Iosif. Space-reduction strategies for model checking dynamic systems. In *Proceedings of the 2003 Workshop on Software Model Checking*, volume 89 number 3 of *Electronic Notes on Theoretical Computer Science*. Elsevier, July 2003.
30. E. Rodríguez, M. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens, and Robby. Extending sequential speciﬁ cation techniques for modular speciﬁ cation and veriﬁ cation of multi-threaded programs. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*. Springer-Verlag, 2005. To appear.
31. E. Rodríguez, M. B. Dwyer, J. Hatcliff, and Robby. A ﬂexible framework for the estimation of coverage metrics in explicit state software model checking. In *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 210–228. Springer, 2004.
32. D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Jan. 1995.
33. SAnToS. SpEx Website. `http://spex.projects.cis.ksu.edu`, 2003.
34. SAnToS. MAnTA Website. `http://manta.projects.cis.ksu.edu`, 2004.
35. S. D. Stoller. Domain partitioning for open reactive systems. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 44–54. ACM Press, 2002.
36. O. Tkachuk, M. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *Proceedings of the 18th International Conference on Automated Software Engineering*, pages 116–129. IEEE Computer Society, October 2003.
37. O. Tkachuk and M. B. Dwyer. Adapting side effects analysis for modular program model checking. In *Proceedings of the Fourth joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Sept. 2003.
38. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer-Verlag, 2001.
39. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the 15th IEEE Conference on Automated Software Engineering*, volume 10 number 2, pages 203–232. Springer, September 2000.
40. J. M. Wing. Writing Larch interface language speciﬁ cations. *ACM Trans. Program. Lang. Syst.*, 9(1):1–24, 1987.