Checking Strong Specifications Using An Extensible Software Model Checking Framework*

Robby, Edwin Rodríguez, Matthew B. Dwyer, and John Hatcliff

Department of Computing and Information Sciences Kansas State University **

Abstract. The use of assertions to express correctness properties of programs is growing in practice. Assertions provide a form of checkable redundancy that can be very effective in finding defects in programs and in guiding developers to the cause of a defect. 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 language for Java, can be validated using a customized 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 of model checking relative to other specification validation techniques and present data that suggest that the cost of model checking strong program specifications is practical for several real programs.

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 [6]. Those foundational ideas inspired the development of more elaborate design practices and methodologies, for example, design-by-contract [15]. 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 [21]. The increasing numbers of modern languages (e.g., Java, C#, PHP) and implementation frameworks (e.g., .NET, MFC) that include simple assertion mechanisms suggests that they are poised to finally having 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 only reasons about the individual program behaviors that are executed. This

* 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.
{robby,edwin,dwyer,hatcliff}@cis.ksu.edu

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 the program 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 null-ness and array bounds [5] and propositional temporal properties [25]. 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 [13]. 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" paradigm by including notation for pre/post-conditions and invariants. JML uses Java's expression syntax and adds, for example, constructs for quantification over object instances and for expressing detailed properties of heap allocated data. This allows developers to create very natural and compact statements of strong specifications of the behavior of Java programs.

In this paper, we describe how we have adapted a flexible model checking framework called Bogor [18] 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.

Previous work on using model checking to verify stronger specification has achieved only limited success for several reasons. First, existing model checkers, such as Spin [9], do not provide direct support for modeling dynamically allocated objects and heap structures making it difficult to even represent the program's behavior; Bogor maintains an explicit, yet compact, representation of the dynamic program heap [20]. 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 [3]. 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 via user-defined atomic expressions that can be evaluated over the full extent of a program state including the heap [18].

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;

Tool (technology)	Automation Usability	JML Coverage	Behavior Coverage	Scalability
LOOP[24] (semi-automated theorem proving)	fair (straight line code), poor (otherwise)	very high	complete (for sequential)	poor
ESC[5] (automated decision procedures)	good (annotations usually needed)	low	high (for sequential), moderate (otherwise)	excellent (modular treatment of methods)
JMLC[2] (run-time monitoring)	excellent	moderate	low (determined by test harness)	excellent
Bogor[18] (model checking)	excellent	very high	moderate (determined by test harness)	good (for unit-level reasoning)

Table 1. JML Reasoning Tools and Technologies

– 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, in most cases, be eliminated through the use of sophisticated state-space reductions.

In the next section, we survey existing technologies and tools for reasoning about JML specifications; we also discuss non-JML based approaches. Section 3 introduces a JML annotated Java example that will be used to illustrate the analysis techniques we have developed. 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.

2 Background

JML is emerging as a popular assertion definition language for Java. It is a rich specification language that can support a range of uses from simple assertions about method parameters and local data to a complete design-by-contract methodology with abstract behavioral interface specifications. Burdy et. al. [1] survey the steadily growing body of tool support for reasoning about JML specifications. Broadly speaking, there are three underlying technologies used in these tools: semi-automated theorem proving, automated decision procedures (a restricted form of theorem proving), 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?

Table 1 summarizes the strengths and weaknesses of the basic technologies in terms of these dimensions. We cite specific tools that implement JML reasoning with those technologies, but the strengths and weaknesses mentioned are, for the most part, characteristics of the underlying technology. We note that despite their weaknesses each of these tools is *useful* in that they have been used to find errors in real Java programs.

LOOP [24] 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 [16]. The semantics of the code and specifications are represented as PVS theories, and users verify specifications against the 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. While recent advances in LOOP's calculus allow nearly automatic verification of methods with straight-line code performing integer calculations, for general Java applications, LOOP scales poorly 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 is another theorem-prover-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 theorem prover. It gains a high degree of automation by treating a small subset of JML and by sacrificing precision 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. 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).

Cheon and Leavens [2] 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 pre-condition state values in post-conditions. `jmlc` implements run-time checking on top of existing JVMs and consequently it provides no direct support for multi-threaded programs.

2.1 Other Related Work

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 most of the existing work on simple assertion languages and manual formal methods is lacking. Recent work on OCL (Object Constraint Language) and Alloy are 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 (cf. [10]) is similar to `jmlc` in that it generates run-time assertions for checking Java. Another direction is

to compare OCL specifications with other UML models (cf. [17]) rather than program source code. However, a number of the issues regarding reasoning about heap-allocated program data are not considered in that work.

The Alloy Annotation Language (AAL) [11] 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.

2.2 JML Model Checking

The work described in this paper complements existing JML tools by model checking. As summarized in Table 1, our tool 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.

3 An Example

Figure 1 presents a concurrent linked-list-based queue from [12] with some JML specification that we have added to describe its behavior. Instances of the `LinkedNode` class implement the nodes of the linked list representing the queue. The `LinkedQueue` class provides `put` and `get` (not shown) methods that implement a fine-grain locking protocol, through the use of 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 pre/post-condition behavior of that functional code we have re-factored it into additional protected methods, for example `insert2`.

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 node forms the head of every list). Whenever a thread attempts to `get` an object from an empty queue, the thread is blocked (the code is not shown). If the queue is not empty, then only the head is locked, and its stored value is returned. The dequeueing 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 written in Java comments with special tags such as `//@`. Pre-conditions and post-conditions for non-exceptional return are written using the JML keywords [`requires`] and [`ensures`] respectively. The [`non_null`] annotation on a reference-type field of an object o is an invariant that the field never has a null value. General invariants on instance data are stated using [`instance invariant`] clauses. The instance invariants for a class C (as well as invariant short-hands such

```
public class LinkedQueue {
 final /*@ non_null @*/ Object putLock;
 /*@ non_null @*/ LinkedNode head;
 /*@ non_null @*/ LinkedNode last;
 int waitingForTake = 0; ...
 //@ invariant waitingForTake >= 0;
 //@ 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; @*/
 void insert2(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);
 }
 synchronized Object extract() {
  synchronized (head) return extract2();
 }
 /*@ behavior
   @  assignable head, head.next.value;
   @  ensures \result == null
   @   || (\exists LinkedNode n;
   @       \old(\reach(head)).has(n);
   @       n.value == \result
   @       && !(\reach(head).has(n))); @*/
 Object extract2() {
  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); @*/
 void insert(Object x) {
  synchronized (putLock) {
   LinkedNode p = new LinkedNode(x);
   synchronized (last) insert2(p);
   if (waitingForTake > 0)
    putLock.notify();
   return;
  } }
}
class LinkedNode {
 public Object value;
 public LinkedNode next; ...
 /*@ behavior ensures value == x &&
   @                next == null; @*/
 public LinkedNode(Object x) {
  value = x;
 }
}
```

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

as [non_null]) are required to hold true in special states that JML defines as *visible states*. The actual definition is somewhat involved, but the basic idea is that the invariant is not required to hold before the object is initialized nor during intermediate steps that occur in methods of C.

The [assignable] clauses state a form of *frame condition* for a method: only the variables listed in [assignable] are allowed to be assigned to. However, locations that are local to the method (or methods it calls) and locations that are created during the method's execution are not subject to this restriction.

The isEmpty method's post-condition states that the method returns true if and only if there is only one node in the list (i.e., the list header). For the extract2 method, the post-condition states that the result is null (when the list is empty) or that there exists a node n such that n is in the list in the pre-state of the method, n is returned as the method result, and n is not in the list in the post-state of the method. The construct

[\old(e)] refers to the value that the expression e had in the pre-state of a method, and [\reach(x)] gives the set of objects reachable by following reference chains originating from x. The `put` method illustrates a *heavyweight* specification where all possible invocations are treated by one of the [`behavior`] clauses. For the `insert` method, a pre-condition requires that the argument giving the object to be inserted is not null, and the post-condition ensures that the last list node holds the object supplied for insertion. The [\fresh($x_1, \ldots, x_n$)] construct specifies that x_i is non-null, and the object pointed to by x_i was not allocated in the pre-state of m.

JML is a large and complex specification language and space constraints make it impossible to detail all of the language features. 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 [19].

4 Checking JML Specifications with Bogor

All the JML checking tools of Table 1 have 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.

A significant portion of the effort in implementing JML checking is associated with the translation phase. This is non-trivial, since it is this phase that captures the JML semantics associated with class inheritance, method overriding, etc. 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 a previously defined method, is a combination of all the pre-conditions 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. In addition, since invariants are checked at method entry/exit, invariants are conjoined with pre/post-conditions to form the effective pre/post-conditions. Fortunately, the JML definition is reasonably clear about the rules for forming the structure of effective pre/post-conditions [13]. Of the JML tools described earlier, our implementation architecture is most closely related to that of `jmlc` since it and `jmlc` both translate to executable representations (bytecode and Bogor models, respectively).

The contrasts that we draw with `jmlc` stem from the fact that using Bogor as a verification engine provides significant flexibility. The target representations produced by `jmlc` have a fixed granularity of actions (bytecode plus assertions), and `jmlc` has no control over the execution of those actions (they are simply executed by a normal JVM). On the other hand, one can think of Bogor as an extensible interpreter where richer verification primitives (e.g., quantification over heap structures) can be implemented directly using Bogor's extension mechanisms, and where direct control over action execution (e.g., scheduling of thread actions) can be obtained using Bogor's pluggable state-space exploration engine modules.

In the rest of this section, we give an overview of how we use the flexibility of Bogor to implement almost all of the JML language. We do not discuss the details of the translation process to Bogor, since general strategies for translating JML have already been described in other work [2]. Our support for JML is made possible by several novel capabilities of the Bogor model checking framework.

Richer verification primitives: `jmlc` must represent all verification requirements as regular Java bytecode. With Bogor, we add primitives to the modeling language to directly represent almost all JML constructs such as quantification, [`\reach`], [`\old`], etc. Many of these constructs are very difficult to represent using Java bytecode/assertions and almost impossible to represent correctly in the presence of concurrency. 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. For correctness in the presence of concurrency, all these objects should be locked in a single atomic step to prevent other threads with direct access to those objects from modifying them during the evaluation of the quantification expression, but this is impossible with Java bytecodes.

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 traversing the Bogor representation of the current state.

Direct access to state history: Fully implementing [`\old`] using only bytecode/assertions is virtually impossible since in general the state of all objects reachable from the argument of [`\old`] must be preserved (this is addressed in detail below). Since model checkers naturally save a compact representation of each encountered state, [`\old`] can be easily implemented by calling the state-space management facilities in Bogor to retrieve relevant portions of the pre-state.

Control of interleaving: In the presence of concurrency, it is difficult to implement checking of almost all JML pre/post-conditions or invariants using bytecode/assertions since, conceptually, evaluations of these expressions should happen in a single atomic step (i.e., there should be no interference from other threads). There are a number of problems in trying to achieve this by 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, and other threads can simply be suspended during the evaluation of a specification expression – this 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, giving the relatively high behavior coverage referenced in Table 1.

4.1 Lightweight versus Heavyweight Specifications

One can write partial JML specifications that capture correctness properties of a fragment of code in a manner similar to the use of assertion facilities. JML also allows users to enrich the specification of parts of a program even to the point of giving a

complete specification of total correctness for a method or class. These heavyweight specifications are distinguished in JML by the use of the [`behavior`] keyword. Users specify the different cases of a method's intended behavior using separate [`behavior`] clauses. A heavyweight specification should be *complete* (i.e., each possible invocation context satisfies the [`requires`] part of a [`behavior`] clause) and *deterministic* (i.e., no invocation context satisfies the [`requires`] part of more than one [`behavior`] clauses). Given the expressiveness of JML it is not always easy to determine whether these constraints are met, so our model checking framework checks for completeness and determinism of heavyweight JML specifications.

4.2 Pre/Post-conditions and Invariants

The JML constructs [`requires`,`ensures`,`signals`] are used to specify pre-conditions, normal post-conditions, and exceptional post-conditions, respectively. Normal post-conditions are checked on method exits caused by executing a `return` bytecode in Java, and exceptional post-conditions are checked on exits caused by an uncaught exception. As described earlier, we check pre-conditions for a thread t entering a method m when t's PC (program counter) is at (before executing) the first bytecode instruction of m. The Bogor representation of the pre-condition is wrapped together with the representation of the first bytecode of the method in a Bogor atomic block, which guarantees that no interleavings can occur between the start of the checking and the completion of the first bytecode. For normal post-conditions the following actions are wrapped in a Bogor atomic block: the return expression (if it exists) is evaluated and the resulting value is assigned to a temporary variable, the post-condition is evaluated (occurrences of `\return` yield the value held in the temporary variable), and the return control action is executed. Without such support (e.g., [2]), 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 post-condition is evaluated. For exceptional post-conditions, 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, the assertion is checked, and then the exception is re-thrown.

A JML [`invariant`] is checked in "visible states" as described in Section 3. This means that the notion of invariant in JML is weaker than the notion of invariant used in model checking where invariants are required to hold in *every* state.

4.3 Referencing Pre-states

[`\old`(e)] yields the value of the expression e evaluated in the pre-state of a method 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. Run-time checking of [`\old`] appearing in a post-condition p can be implemented by (a) storing the value of e in a special local variable v_e when entering m and then (b) replacing `\old(e)` with v_e in p [2]. When [`\old`] expressions involve object references, especially comparisons between method pre- and post-state references, this approach may require storage of large portions of the pre-state heap. Despite the potential costs involved, supporting reference values

in [\old] is necessary if one aims to express strong properties about heap data. In a concurrent setting, an additional complication arises if there are *multiple* pre-states associated with a particular post-state 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. A model checker explores all interleavings of threads, and therefore it can naturally check a post-condition with respect to all associated pre-states.

Since an explicit-state model checker such as Bogor stores all states, one would think that we can simply retrieve appropriate pre-states from the model checker's depth-first-stack of visited states when evaluating [\old]. Unfortunately, this straightforward strategy may miss some error states because the model checker may end up hitting a state stored in the cache (and thus backtrack) before it reaches the exit points of a method (and this may happen even though the pre-states are different).

Figure 2 presents a fragment of Java with a simple post-condition and the state-space constructed by Bogor using a depth-first search state exploration with two instances of the `Race` thread. For simplicity, a state is denoted by a vector comprising 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 second instance of `Race`. We use $\bullet$ to denote a thread that has died or has not been created yet. We denote the PC of threads at loc$X$ by the integer X. The first location of the main thread is `loc0`. 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 resulting state (i.e., the resulting state is stored in the model checker's cache). In order to reduce the state-space, we use the thread symmetry reduction presented in [20]. This causes, for example, the state $\langle 0, \bullet, 1, 2\rangle$ to be considered as observationally equivalent to the state $\langle 0, \bullet, 2, 1\rangle$. We also use the partial-order reduction presented in [3]. This causes all the transitions of the main thread (not shown – it simply creates the two instances of `Race`) to execute without any interleavings of the newly created `Race` instances. The reductions do not affect the result of checking the post-condition; the problem that we are presenting also occurs in the unreduced state-space. Also note that the post-condition is checked whenever `loc4` is executed. That is, the execution of the `return` statement is aggregated with the transition that checks the post-condition in an atomic transition.

As can be observed, there exists a trace through the state-space of Figure 2 that violates the post-condition:

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

Specifically, at step $\langle 1,\bullet,\bullet,4\rangle \dashrightarrow \langle 1,\bullet,\bullet,\bullet\rangle$ the post-condition will fail because the value of `x` at one of the pre-states of the second instance of `Race` (i.e., $\langle 1,\bullet,4,3\rangle$) 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 because it has seen the state $\langle 1,\bullet,4,3\rangle$ from a different trace. Thus, the subsequent steps (including the post-condition check) in the error trace are not encountered in the state-space exploration.

We solve this problem by identifying a portion of the pre-state that can be used to distinguish the post-states; it suffices to consider the set of objects reachable from references that are visible in the pre-state. This calculation can be performed efficiently

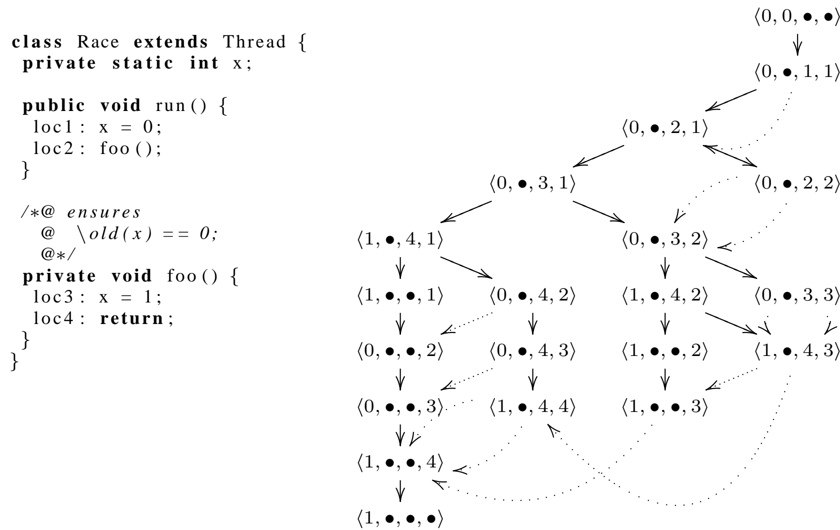

Fig. 2. Race Example and its DFS state-space

in Bogor because: (1) Bogor employs state-of-the-art collapse compression that reuses parts of previous states when storing a new state [20], and (2) we can augment the thread state that will execute the post-conditions containing [`\old`(e)] by the collapsed state encoding the relevant pre-state objects. The result is similar to adding the collapsed pre-state as a local variable of the method, for example,

```
private void foo() {
  int collapseState = 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 post-states with different pre-states distinguishable from each other (i.e., observationally inequivalent). In general, this addition to the state space might cause significant increase in checking time and space, but as we show in Section 5, this can be mitigated through the use of reduction techniques that detect and exploit atomic method execution as determined by partial order reduction [8].

4.4 Methods in JML Expressions

JML specification expressions can 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 (described below) is empty. Under this definition, synchronized methods, such as `isEmpty` in Figure 1, are not pure since they require modification of an object’s lock state; it would be useful

to consider them as pure as discussed in [14]. To address this, users can configure the model checker to require only that annotated methods are *weakly pure*. A weakly pure method can contain assignments (e.g., to local variables) if the state observed by other threads does not change. In other words, in a context in which the method is executed without any interleavings, the post-state of the method should be identical to its pre-state (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 pre-states and post-states of methods as they are called.

4.5 Object Operations

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. In JML, 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 precisely due to the presence of aliasing, and consequently many tools simply avoid this check. Bogor's explicit representation of the heap can be used to exactly determine variable aliasing and to 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 recorded is flagged as an error. In addition, for nested method calls the semantics of [`assignable`] requires that the sets of assignable locations of a nested method are a subset of those for any enclosing method. Bogor can easily check this on-the-fly since its explicit heap representation keeps precise alias information.

[`\reach`(x)] gives the objects reachable by following reference chains originating from x. JML also includes variants of [`\reach`] that filter the objects based on their types and field navigations [13]. Heap reachability is used in Bogor for partial-order reductions [3] and thread symmetry reduction [20]. Given this existing functionality in Bogor, [`\reach`(x)] is easily evaluated by calling the appropriate Bogor libraries.

[`\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 [3], and it can be implemented by calling the existing Bogor libraries, just as with [`\reach`(x)].

[`\fresh`($x_1, \ldots, x_n$)] requires that, at the post-states of a method m, variables x_i are non-null and the objects bound to x_i are not present in any of the pre-states of m. [`\fresh`] implicitly accesses the pre-state of a method, thus, we adapt the strategy of storing additional data to distinguish pre-states that was developed for [`\old`]. For [`\fresh`], however, we explicitly store newly allocated object references in a local variable to minimize the stored state information, since the number of allocated objects in a method activation is usually smaller than the set of all objects in the pre-state.

4.6 Logic Operations

The universal quantification expression [`\forall`(τ 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 τ

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 reachable τ objects from any global variables or threads. The existential quantification expression [`\exists`$(\tau\ X; R(X); C(X))$] is supported similarly as [`\forall`].

The set comprehension notation expression [`new JMLObjectSet` $\{ \tau\ x \mid P(x) \}$] gives the set of values of type τ that satisfies the predicate P. We model the abstract set using the generic set type extension described in [18]. The set construction is done using a similar approach as [`\forall`].

5 Evaluation

Support for JML features has been added to Bogor through its language extension facilities [18]. We extended Bogor's input language syntax with JML's primitive operations and implemented their semantics by using Bogor's APIs [19] to access the full program state and the trace of states leading to the current state.

We used Bogor to reason about six real Java programs, five 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- and post-conditions 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 (s) and memory in mega-bytes; data was gathered running Bogor under JDK 1.4.1 on a 2 GHz Opteron (32-bit mode) with maximum heap of 1 GB running Linux (64-bit mode).

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 pre-state heap must be used to distinguish post-states. Comparing runs with and without POR reveals the significant benefit of sophisticated POR; it yields between 2 and 4 orders 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.

Program	POR and JML	no JML	no POR	no JML and no POR
BoundedBuffer[7]	164 loc	\fresh, \old, signals		
3 threads	69 states	69 states	2647 states	2647 states
10 objects	1 s/0.8 MB	1 s/0.6 MB	4 s/1.2 MB	3 s/1.0 MB
7 threads	1098 states	1098 states	1601745 states	1601745 states
18 objects	26 s/1.2 MB	23 s/1.0 MB	8936 s/180.2 MB	8458 s/167.7 MB
DiningPhilosopers[7]	193 loc	\forall, \fresh		
4 threads	38 states	38 states	12514 states	12514 states
6 objects	1 s/1.1 MB	1 s/0.7 MB	27 s/2.5 MB	20 s/2.0 MB
6 threads	1712 states	1712 states	1939794 states	1939794 states
8 objects	32 s/2.3 MB	24 s/1.8 MB	9571 s/159.9 MB	8719 s/157.6 MB
LinkedQueue[12]	228 loc	\fresh, \reach, \old, signals, \exists		
3 threads	2833 states	1533 states	17064 states	11594 states
22 objects	10 s/1.6 MB	5 s/1.0 MB	38 s/3.7 MB	21 s/2.3 MB
5 threads	39050 states	12807 states	1364007 states	423538 states
32 objects	144 s/5.9 MB	72 s/2.5 MB	14557 s/140.5 MB	2415 s/46.4 MB
RWVSN[12]	227 loc	\old		
4 threads	183 states	183 states	2621 states	2255 states
5 objects	1 s/1.0 MB	1 s/0.8 MB	2 s/1.5 MB	2 s/1.0 MB
7 threads	18398 states	18398 states	4995560 states	4204332 states
9 objects	185 s/6.8 MB	144 s/3.0 MB	34804 s/463.7 MB	26153 s/366.3 MB
ReplicatedWorkers[4]	543 loc	\fresh, \old, \reach		
4 threads	1751 states	1751 states	322016 states	269593 states
19 objects	14 s/2.1 MB	13 s/1.9 MB	897 s/29.8 MB	716 s/26.6 MB
6 threads	10154 states	10154 states	12347415 states	10016554 states
21 objects	99 s/3.3 MB	92 s/2.8 MB	30191 s/391.8 MB	21734 s/282.5 MB
Arrays.sort(Object[])	151 loc	\forall, \exists, \old		
1 thread	2 states	2 states	21597 states	21597 states
502 objects	82 s/2.0 MB	7 s/1.9 MB	391 s/49.5 MB	343 s/48.8 MB

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

5.1 Discussion

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 counter-examples proved to be extremely useful in debugging erroneous specifications. The exhaustive nature of the search in model checking makes it a much more effective *specification debugging* tool than run-time checking.

We included a standard comparison sorting program in our set of examples to demonstrate Bogor's behavior on a declarative JML specification of rich behavioral properties (i.e., the post-state is an ordered permutation of the pre-state). Despite the richness of this specification, due to the single-threaded nature of the program the

method trivially executes atomically, and there is no overhead for JML checking. Our partial order reduction 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 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 initial data suggests that the combination of these techniques can provide cost-effective reasoning about non-trivial properties of real 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 input language. We believe this alternative will be more efficient than [2] because we can apply various reduction algorithms to those ADTs, as shown in [18].

In this paper, we considered complete Java programs, but we plan to support the analysis of partial programs as well. Ongoing work [23] is exploring techniques for synthesizing test harnesses for unit level reasoning. 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 [22] towards that end.

References

1. 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*, 2003.
2. Y. Cheon and G. T. Leavens. A runtime assertion checker for the java modeling language. In *Proceedings of the International Conference on Software Engineering Research and Practice*, 2002.
3. M. B. Dwyer, J. Hatcliff, V. R. Prasad, and Robby. Exploiting object escape and locking information in partial order reductions for concurrent object-oriented programs. *Formal Methods in System Designs*, 2004. (to appear).
4. M. B. Dwyer and V. Wallentine. A framework for parallel adaptive grid simulations. *Concurrency : Practice and Experience*, 9(11):1293–1310, Nov. 1997.

5. 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*, 2002.
6. R. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Mathematics*, 1967.
7. S. Hartley. *Concurrent Programming - The Java Programming Language*. Oxford University Press, 1998.
8. J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model checking. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, Jan. 2004.
9. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
10. H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In *The Third International Conference on The Unified Modeling Language (LNCS 1939)*, 2000.
11. 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 applications*, 2002.
12. D. Lea. *Concurrent Programming in Java: Second Edition*. Addison-Wesley, 2000.
13. G. T. Leavens, A. L. Baker, and C. Ruby. JML: a Java modeling language. In *Formal Underpinnings of Java Workshop*, Oct. 1998.
14. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In *Proceedings of the 1st International Symposium on Formal Methods for Components and Objects*, Nov. 2002.
15. B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.
16. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (LNCS 607)*, 1992.
17. M. Richters and M. Gogolla. Validating UML models and OCL constraints. In *The Third International Conference on The Unified Modeling Language (LNCS 1939)*, 2000.
18. 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*, 2003.
19. Robby, M. B. Dwyer, and J. Hatcliff. Bogor Website. `http://bogor.projects.cis.ksu.edu`, 2003.
20. 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*, July 2003.
21. D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Jan. 1995.
22. S. D. Stoller. Domain partitioning for open reactive systems. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2002.
23. 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*, Oct. 2003.
24. 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 (LNCS 2031)*, 2001.
25. W. Visser, K. Havelund, G. Brat, , and S. Park. Model checking programs. In *Proceedings of the 15th IEEE Conference on Automated Software Engineering*, Sept. 2000.