

Checking JML Specifications Using An Extensible Software Model Checking Framework[★]

Robby¹, Edwin Rodríguez¹, Matthew B. Dwyer², John Hatcliff¹

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

August 5, 2004

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 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

* 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 Software, 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.

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 [7]. Those foundational ideas inspired the development of more elaborate design practices and methodologies, for example, design-by-contract [18]. 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 [27]. The increasing numbers of modern languages (e.g., Java, C#, PHP) and implementation frameworks (e.g., 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 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 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 [6] and propositional temporal properties [31]. 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 [15]. 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 [18] 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 [23] 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 [11], 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 [25]. 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 [23].

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 Bogor and 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 and Related Work

2.1 Bogor

Bogor [23] is an extensible and highly modular software model checking framework that can be adapted and customized to the specific characteristics of different problem domains. For example, Bogor has been customized to exploit domain-specific features in the verification of avionics systems [4].

One of the features that makes Bogor flexible is the extensibility of its modeling language, BIR [23]. In contrast to other model checker input languages, 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 essentially be used to build abstract machines tailored to specific application domains. In additions, many optimizations can be done in BIR language extensions such as removing unnecessary details from ADT bit-vector state representations and symmetry reductions.

Figure 1 shows the mechanism for extending the BIR language. The figure gives a glance at the BIR syntax displaying a simple system that models an environment in which there are several processes competing for resources in a resource 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

```

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>(resourcePool)
    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. 1. Resource Contention Example

for keyword provides the fully qualified name of the class that implements the extension. The implementation of the extension operations is done by following a clean and well defined API using widely known design patterns [8]. These language extension facility is used as a vehicle to define JML constructs directly in BIR; Those extensions are presented in Section 4.

Another important form of Bogor extension is module extension. Figure 2 shows a sketch of Bogor's architecture. The components in the *Model Checking Components* area of the figure, are loosely coupled and communicate with each other through well defined interfaces. They are all loaded dynamically by reading the location of their implementing classes from a user defined configuration file. Therefore, changing the framework behavior is as easy as providing a different implementation for a given module that is customized to a specific domain. This second type of extension is needed to implement JML operators, especially those that need to inspect the state of the heap and access history information.

2.2 JML: The Java Modeling Language

The Java Modeling Language (JML) [15] is a Behavioral Interface Specification Language (BISL) [32] designed at Iowa State University by Gary Leavens and others. In JML, there are two ways one can specify properties of programs, i.e., by using *lightweight specifications* or *heavyweight specifications*. The difference between heavyweight and lightweight specifications is on how methods are annotated (independently of class level annotations: invariants, axioms, etc.). A specification is heavyweight as long as it is annotated using one of the *behavior* keywords (*behavior*, *normal_behavior* or *exceptional_behavior*), otherwise the specification is lightweight. In JML, the interface of a method is specified using a set of different clauses, each of which represent a different aspect of the behavior of the method. The most important clauses are:

- *requires*: Used to specify the conditions that the callers of this method must satisfy.
- *ensures*: Specifies the conditions that this method must guarantee to its callers.

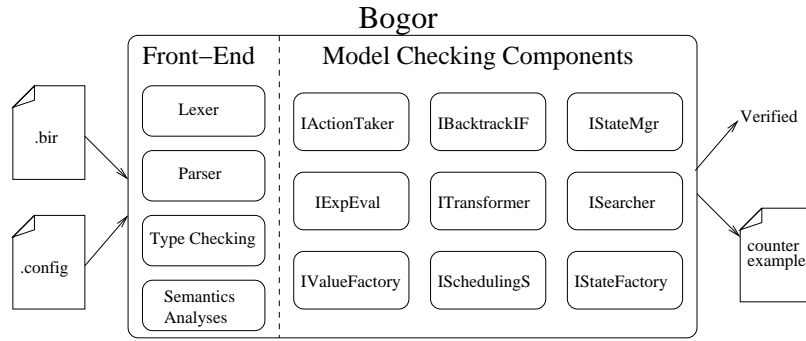


Fig. 2. Bogor Architecture.

- `diverges`: Specifies a condition that, when true, makes this method diverge, i.e., never return to its caller.
- `assignable`: Used to specify frame conditions by providing a list of memory locations that can be modified by this method.
- `when`: Specifies an enabling condition: when called, this method should only proceed execution if the condition specified is true, otherwise, it should block until it is true.
- `signals`: Used to specify exceptional behaviors. An exception type and a condition is given. The semantics is that, if an exception of the given type is thrown, then the condition must be true.

Heavyweight specifications are called such because they are assumed to be complete: if the specifier omits any clause, the omitted clause is given a default value. In contrast, in lightweight specifications, nothing is assumed about omitted clauses. Table 1 displays this situation.

One interesting consequence of Table 1 is that heavyweight specifications, by default, always provide a description of total correctness: the method must always terminate (i.e., its `diverges` clause is true by default). On the other hand, in lightweight specifications, unless explicitly stated, nothing is assumed about termination, therefore it provides a description of partial correctness.

One of the main features that has made JML appealing is its Java-like syntax. In fact, the JML specs can exploit Java code, for example, by calling Java methods from within a specification. Also, JML provides a rich library of *model classes* that can be used to construct rich abstract descriptions of program behavior. For example, the library includes a set of data structure model classes that can be used to model abstract properties of concrete data structures, in a more concise way, without unnecessary implementation details.

In addition, there is also a rich set of native operators for defining complex specifications, among the most important are:

- `\old(e)`: used only in the specification of postconditions and allows one to access the value of expression `e` in the prestate of the method.
- `\reach(e)`: returns a set with all the objects that are reachable from `e` (`e` must evaluate to a reference value).
- `\forallall`, `\exists`: allows one to state properties using logical quantification, specially useful to quantify over all or some of the objects in the heap.

These characteristics are driving JML as an emerging standard assertion definition language for Java.

2.3 Tool Support for JML Verification

Burdy et. al. [1] 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?

Table 2 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,

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

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

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 [30] 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 [20]. Thus, the semantics of the Java code as well as JML specifications are represented as PVS theories, and users verify the 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 weak-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 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. It uses a modular checking approach in which methods are verified in isolation by trying to prove that class invariants and method post-conditions hold under the assumption that the method’s pre-conditions 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).

Cheon and Leavens [2] have developed a run-time checker (`jmlc`) which compiles JML-annotated programs into byte-code 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.

2.4 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 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., [12], 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., [22], 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) [13] 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 invoca-

Tool (technology)	Automation Usability	JML Coverage	Behavior Coverage	Scalability
LOOP[30] (semi-automated theorem proving)	fair (straight line code), poor (otherwise)	very high	complete (for sequential)	poor
ESC[6] (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[23] (model checking)	excellent	very high	moderate (determined by test harness)	good (for unit-level reasoning)

Table 2. JML Reasoning Tools and Technologies

tions. 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.

2.5 JML Model Checking

The work described in this paper complements existing JML tools by model checking programs written in multi-threaded Java against specifications using almost all of the features of JML. Existing work on model checking has not supported rich specification languages like JML because existing model checkers such as SPIN or SMV (a) do not provide direct support for dynamically created objects and inheritance, (b) they do not provide state-space representations or exploration strategies that are optimized for dynamic object-oriented programs which are crucial for reducing the costs of software model checking, and (c) they do not provide extension facilities that allow that complex state predicates involving object instance quantification or heap-reachability properties to be easily encoded. As summarized in Table 2, 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

In this section we introduce an example that will be used throughout the rest of this paper. Figure 3 presents a concurrent linked-list-based queue from [14] with some JML specifications that we have added to describe its behavior. Instances of the class `LinkedListNode` implement the nodes of the linked list representing the queue. The `LinkedListQueue` class provides `put` and `get` (not shown) 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 pre/post condition behavior of that functional code we have refactored it 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 node with an unused data field 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 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 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]`

```

class ListNode {
public Object value;
public ListNode next;

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

public class LinkedQueue {
protected final /*@ non_null @*/ Object putLock;
protected /*@ non_null @*/ ListNode head;
protected /*@ non_null @*/ ListNode 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 ListNode(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(ListNode 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 || (\existss ListNode n;
@ \old(\reach(head)).has(n);
@ n.value == \result
@ && !(\reach(head).has(n)));
@*/
protected Object refactoredExtract() {
Object x = null;
ListNode 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) {
ListNode p = new ListNode(x);
synchronized (last) refactoredInsert(p);
if (waitingForTake > 0) putLock.notify();
return;
}
}

```

Fig. 3. A Concurrent Linked-list-based Queue Example (excerpts)

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 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 . Accordingly, visible states include those at the end

of execution of a constructor for C , and those at the entrance and exit of methods calls when an instance of class C is the receiver, and states where no constructor or instance method for C is in progress. This last condition ensures that changes to a public field that do not occur through methods of C are visible to the invariant.

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 the methods that it calls) and locations

that are created during the method’s execution are not subject to this restriction.

For the `isEmpty` method, the post-condition states that the method returns true if and only if there is only one node in the list (i.e. the dummy node always at the head of the list). For the `refactoredExtract` 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 $[\text{old}(e)]$ refers to the value that the expression e had in the pre-state of a method, and $[\text{reach}(x)]$ gives the 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 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 $[\text{fresh}(x_1, \dots, x_n)]$ 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 do not allow for detailed descriptions of all of its 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 [26].

4 Checking JML Specifications with Bogor

All the JML checking tools of Table 2 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.

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. 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 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 [15]. Of the JML tools described earlier, our implementation architecture is the most closely related to that of `jmlc` since 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 the verification of a rich subset of the JML language. 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, $[\text{reach}]$, $[\text{old}]$, etc. Many of these constructs are 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 to achieve using Java bytecodes without modeling the Java Virtual Machine.

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: 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, 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 – giving the relatively high behavior coverage referenced in Table 2.

4.1 Coverage of JML Annotations

A consequence of the flexibility provided by Bogor in verifying JML specifications, is that we have achieved a high coverage of JML language in terms of the number of annotations handled and also of the complexity of the annotations that can be verified. Of all the JML features covered, the most noteworthy are:

- **invariants:** Bogor is able to check class invariants on all objects present in the heap, by means of a relaxation that makes the invariants verification process more efficient (Section 4.4).
- `\old()`: this is an important operator in JML since it allows accessing history states and it is fully supported in Bogor.
- **assignable:** another important construct since it allows the specification of frame conditions for a given method that is fully supported in Bogor. Most of the tools that

check this feature do so by using static analysis. Unfortunately, due to the aliasing problem, the results given by such tools is, at best, a good approximation. In Section 4.6 we will see how Bogor is able to give exact information about heap state changes.

Bogor is also capable of checking exceptional code execution because BIR has a built in exception handling mechanism similar to Java. If appropriate environment that simulate exceptional conditions are provided, then Bogor can verify the correctness of exceptional specifications and code.

All of the points discussed above make Bogor one of the JML verification tools with higher coverage of the JML language, not only in terms of the number of annotations handled, but more importantly in the complexity of the specifications that can be checked.

4.2 Translating JML to BIR

In this section, we explain the details of how JML annotations are translated to BIR. Figure 4 shows the BIR translation of the method `refactoredExtract()` of the linked queue program shown in Figure 3. 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. We refer the reader to [26] for more details on this and other examples.

Most of the JML annotations are translated to BIR assertions. Some others are translated to specific operators that have been added to extensions and that usually change, in some way, the model checker behavior. The assertions are normally inserted in every method’s pre and post-state, 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 post-condition, 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 doesn’t 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 is before the check was performed. For example, we can see in Figure 4 how all the checks for the precondition and invariants

```

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: live { }
    do invisible {
    }
    goto locSpec1;
  loc locSpec1: live { [ | r1 | ], [ | r0 | ], collapsedState }
  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.waitForTake | \ >= 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: live { [ | r1 | ], collapsedState }
  do invisible {
    // Last method's instruction
    [ | r0 | ] ./ | linkedqueue.LinkedQueue.head | \ := [ | r2 | ];

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

    // Post-Condition
    spec1 := State.preVal < Set.type < ( | java.lang.Object | ) > >
      ( State.reachSet < ( | java.lang.Object | ) > ( [ | r0 | ] ./ | linkedqueue.LinkedQueue.head | \ ),
        State.currentThreadId ( ) );
    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.waitForTake | \ >= 0 );
    assert ( Set.contains < ( | java.lang.Object | ) >
      ( State.reachSet < ( | java.lang.Object | ) > ( [ | r0 | ] ./ | linkedqueue.LinkedQueue.head | \ ),
        [ | r0 | ] ./ | linkedqueue.LinkedQueue.last | \ ) );
  }
  goto loc13;
  loc loc13: live { [ | r1 | ] }
  do {
  }
  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 `LinkedList` example

have been grouped in a single location: `locSpec1` (In BIR, all the instructions in the same location are executed atomically, without any interleaving).

Another interesting aspect of the translation is the consideration of inheritance. JML method specifications and class invariants are inherited by subclasses, therefore a special treatment must be made to add the appropriate checks in each method. We follow the standard JML desugaring procedure that is explained in [21]. For example, the pre-condition is the disjunction of all the pre-conditions inherited from subclasses, including the local pre-condition. On the other hand, the post-condition is the disjunction of the implications that have as sequent every pre-condition with its associated post-condition as the consequent.

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 4.9).

4.3 Lightweight versus Heavyweight Specifications

In JML, one can write partial specifications that capture specific correctness properties of a fragment of code, thus, JML can be used in much the same way as lightweight assertion facilities (e.g., Java’s `assert` method). Unlike those facilities, JML provides users with the opportunity 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 methods intended behavior using separate `[behavior]` clauses.

Our framework checks both types of specification in a straightforward manner because all specifications are translated as simple assertions. So, the lightweight specifications, which are simple assertions, are inserted in the appropriate position in the BIR code, whereas the heavyweight specifications are more complex, but they translate similarly to a chain of simple assertions in the BIR code.

According to JML definition, a heavyweight specification describes a method’s behavior with respect to *total correctness*, that is, an implicit obligation of a method with a heavyweight specification is termination, unless it is otherwise stated with a `diverges` clause [21]. In a finite system (we restrict our attention to finite state systems, since this is the kind of systems that Bogor can verify), termination can be specified as a liveness property. Therefore, for finite state systems, Bogor is able to perform total correctness verification, according to JML heavyweight specifications definition.

4.4 Checking 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 program counter (PC) is at 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 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. In BIR all instructions in the same location are executed atomically without any interleaving.

For normal post-conditions, 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 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 3) 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, 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 and the class invariants.

A JML `[invariant]` is checked in “visible states” as described in Section 3. 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. Doing this is very expensive. However, an interesting observation is made: throughout the life time of a method’s execution, the portion of the heap that is modified tends to be smaller with respect to the rest of the heap. This is true because most methods are usu-

ally tailored to make local changes upon their target objects. Therefore, when using JML semantics to check invariants, a considerable amount of effort is made in checking conditions on objects that haven't changed their state.

Based on this observation, our framework makes a relaxation when checking invariants: class invariants are only checked in the visible states of methods of the same type as the invariant. Additionally, instance invariants are checked in the visible states of methods that list any field member of objects of the same type as the invariant's in their `assignable` clause. Basically, we try to be smart and check only those objects that are potentially changed. This is a big relaxation with respect to JML's original semantics and relies on proper encapsulation of the classes, that is, objects can only be modified by class methods.

As mentioned before, this simplification is done to reduce the cost of verifying class invariants. However, this approximation comes at some cost. There is the risk of missing some invariant violations in the following situation: a method without an `assignable` clause could break the invariant of an object of a different type because the invariant would not be checked. To address this problem we are planning on adding a static analysis phase in which, for methods that do not have explicit frame conditions, the invariants for the static types of the locations modified in the body of the method are checked.

Figure 4 shows the corresponding translation of invariants for the method `refactoredExtract()`. We recall from Figure 3 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 `LinkedListNode`, which is a class that has no invariants. Hence, in this method we only check the invariants of the class `LinkedListQueue`. The corresponding invariant checks can be seen in Figure 4 at locations `locSpec1` and `locSpec3`. Notice how invariant checks are inserted at the beginning and at the end of the method. The check for each invariant 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 a lot 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). The details of this operator are explained in section 4.8.

4.5 Referencing Pre-states: The `\old` operator

`[\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 can 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 using JML. In a concurrent setting, an additional complication arises since there can be *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, thus, 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 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 5 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, 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 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` as 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 [25]. This causes, for example, the state

```

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;
    }
}
    
```

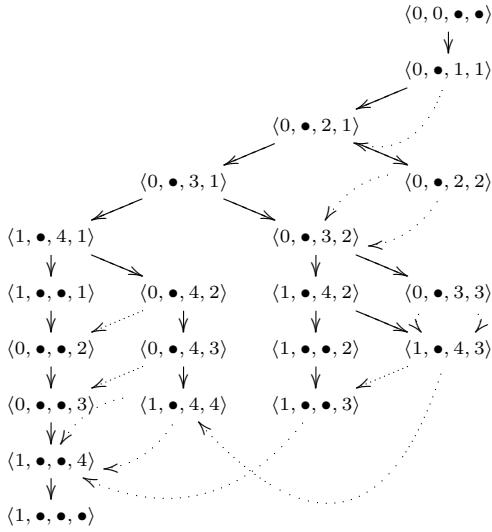


Fig. 5. Race Example and its DFS state-space

$\langle 0, \bullet, 1, 2 \rangle$ 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. Note that 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 5 that violates the post-condition:

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

Specifically, at step $\langle 1, \bullet, \bullet, 4 \rangle \rightarrow \langle 1, \bullet, \bullet, \bullet \rangle$ the post-condition will fail to verify 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 $\langle 0, \bullet, 3, 3 \rangle \rightarrow \langle 1, \bullet, 4, 3 \rangle$ causes `Bogor` to backtrack because it has already 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 in `Bogor` because: (1) `Bogor` employs state-of-the-art collapse compression that reuses parts of previous states when storing a new state [25], and (2) we can augment the thread state that will execute the post-conditions containing $[\backslash\text{old}(e)]$ 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 [10].

Now let us take a look at how we do this in BIR. The post condition of `refactoredExtract()` from Figure 3 is:

```

/*@ ensures \result == null || (\existss 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 pre-state 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 extended BIR with a function that given a reference, it returns the collapsed encoding of the heap reachable from the reference argument. This is what we do in the first instruction of location `locSpec1` in Figure 4. The collapsed heap ID is stored in a local variable and maintained live (BIR locations have a `live` clause, as seen in Figure 4, that spec-

ifies the variables that will be live upon exit of the location¹ up to the check of the post-condition.

The actual calculation of the pre-state value is performed by another extension function called `State.preVal()`, as shown in Figure 4. This value is stored in a temporary variable (`spec1` in this case) and passed along to the property check expression for the post-condition verification. The function `State.preVal()` calculates the pre-state value by using the backtracking facilities of Bogor: starting at the post-state, it backtracks to the pre-state, evaluates the expression, and returns the result of the evaluation.

4.6 Checking Frame Conditions: The assignable clause

The `[assignable ap_1, \dots, ap_n]` method 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 $x.f_1 \dots f_k$, 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 $x.f_1 \dots f_k$, we extend Bogor so that it records that the field f_k of the object represented by $x.f_1 \dots f_{k-1}$ (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 require 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.

¹ The current implementation of Bogor can figure out liveness automatically, so these `live` annotations are no longer necessary.

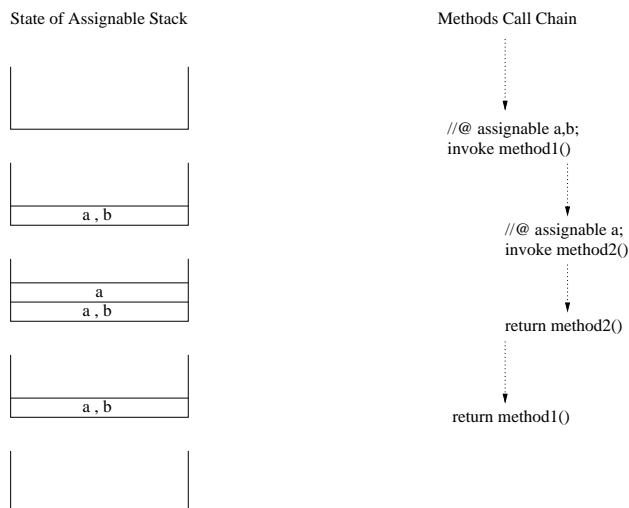


Fig. 6. Mechanism used to check frame conditions in Bogor.

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 that allocated it) 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 reported. This enforces the JML restriction that a method cannot call another method that assigns to locations not listed in its 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<(|LinkedList.LinkedListNode|) > (...);
```

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. This 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 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.

4.7 Methods in JML Expressions

It is convenient to allow JML specification 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 (described below) is empty (nothing). Note that this definition does not allow methods to synchronize on objects as this would represent a modification of an object’s lock state. For example, the `isEmpty` method in Figure 3 cannot be declared as a pure method. However, it would be useful to consider methods such as `isEmpty` as pure as discussed in [16].

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 interleavings, the post-state of the method should be identical to 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. 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.

4.8 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(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 [15]. The basic notion of heap reachability is used extensively in Bogor for partial-order reductions [3] and thread symmetry reduction [25]. Given this existing functionality in Bogor, `[\reach(x)]` is easily evaluated by calling the appropriate Bogor libraries. It just so happens that for performing thread symmetry reduction, Bogor needs to calculate all the objects that are reachable from a given reference (for ordering purposes). Therefore, there is an internal function that, given a reference, returns a set containing all the objects reachable from that reference. This is exactly the semantics of `[\reach()]`, so its implementation reduces to a simple API call.

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

`[\fresh(x1, ..., xn)]` 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 method local 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 pre-state.

4.9 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. but JML has facilities to define both universal and existential quantification.

The universal quantification expression `[\forallall(τ X; R(X); C(X))]` holds true when $C(X)$ is satisfied by all values of quantified variables $X = x_1, \dots, x_n$ 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 the set of reachable τ objects from all global variables and threads.

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

5 Evaluation

Support for JML features has been added to Bogor through its language extension facilities [23]. We extended Bogor’s input language syntax with JML’s primitive operations and implemented their semantics by using Bogor’s API’s [24] to access the full program state and the trace of states leading to the current state.

We applied Bogor to reason about six Java programs, most of which are multi-threaded and manipulate non-trivial heap-allocated data structures. Table 3 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 and memory in mega-bytes of RAM, for each version of the example programs; 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).

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 3 gives details about further configuration of the test drivers: number of threads and number of objects.

We give an overview of the kind of properties verified in each system. In addition, every model was checked for deadlocks, as this is the default check performed by Bogor on any system.

5.1 BoundedBuffer

This program is an implementation of a concurrent buffer, using a fixed size array, obtained from [9]. 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 object 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 in checking frame conditions (`assignable`) and structural invariants, such as checking that the size of the buffer always keeps between bounds.

5.2 DiningPhilosophers

This is an implementation of the dining philosophers problem obtained from [9], with just three classes: `DiningServer`, `Philosopher`, and `DiningPhilosophers`.

The `DiningServer` class can be thought 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, 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 be able to pick up a fork it must be free.

5.3 LinkedQueue

This program has already been described in section 3, so we only talk briefly about some other details in this section. This program has three classes: `LinkedQueue`, `LinkedNode`, and `LinkedQueueDriver`.

Program	POR and JML	no JML	no POR	no JML and no POR
BoundedBuffer[9]	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
DiningPhilosophers[9]	193 loc	\forallforall, \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[14]	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[14]	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[5]	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	\forallforall, \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 3. Checking time/space for JML Annotated Java Programs

The `LinkedQueue` class has already been explained with detail in Section 3. The class `LinkedListNode` 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.

5.4 RWVSN

This program is an implementation of a readers-writers lock obtained from [14]. 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 readers, a group of writers, 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.

5.5 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 [5]. This program has a total of 16 classes, of which we only list the most important ones: `ReplicatedWorkers`, `Coordinator`, `Worker`, and `BasicRWTest`.

The `ReplicatedWorkers` class provides a place to control the whole system: it initializes the workers and the tasks pool, allows to abort the run, collect results, etc. The `Coordinator` class on the other hand, is an entity that coordinates the interaction among the worker threads. The class `Worker` is the class that encloses the task to be done and is controlled by the coordinator. Finally the `BasicRWTest` class is just 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 [5].

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

5.6 *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.

5.7 *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, `LinkedList`, space consumption increased by a factor of three and time by a factor of six. This is not unexpected since the JML specifications for `LinkedList` 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 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.

5.8 *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 counterexamples 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 illustrate Bogor’s behavior on a declarative JML specification of a rich behavioral properties (i.e., the post-state is an ordered permutation of the pre-state). 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 partial order reductions 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 multi-threaded Java programs.

In this paper, we considered complete Java programs, but we plan to support the analysis of partial programs as well, for example, individual classes. Ongoing work [29] 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 [28] towards that end.

7 Future Work

According to Leavens, “JML’s support for concurrency is in its infancy” [17]. Substantial specification forms relevant for concurrent programming have yet to be added to JML. Moreover, we have already noted in previous sections that most JML checking tool do not support reasoning about concurrent programs (e.g., the LOOP tool), or their checking mechanisms are not robust in the presence of concurrency (e.g., run-time checking in `jmlc`). In the following paragraphs, we describe a few of the areas in which JML can be enhanced for the verification of concurrent programs.

We have already described in the previous section how Bogor can easily implement the `[\lockset]` primitive based on the information already collected for Bogor’s partial order reduction strategies [3]. In essence, this construct is used to specify that threads cannot *interfere* when accessing a particular object o because o is locked in a consistent way (e.g., the set of locks held by each thread t when accessing o always contains at least one common lock). The `[\lockset]` form is useful, but there are other common non-interference situations where objects are not protected by locks. For example, a temporary object may be created inside of a method m , but never assigned to anything other than a local variable. Such an object is said to be *method local* because it does not *escape* the method’s activation. Accesses to a method local object o_{ml} need not be lock-protected, because the only references to o_{ml} are held in the activation record for m which is only accessible by one thread (the thread that ran that activation of m). Similarly, it is often the case that an object o will never in its lifetime be reachable from more than one thread at a time. Such objects are said to be *thread local*, and they also do not need to be locked since there will exist no accesses from competing threads that interfere. Finally, a *read only* object o_{ro} may be initialized under the control of one thread, and then shared between threads but with only read accesses occurring. Read-only objects also do not need to be locked to avoid interference.

For developers, it would be useful to be able to specify and have verified that objects are lock-protected, method-local, thread-local, and read-only because this means that the developer does not have to reason about errors being introduced by unanticipated interleavings or race conditions. In fact, it is important for an optimized model checker to know such properties of objects as well. If an object satisfies one of the conditions above, the model checker can avoid exploring some thread interleavings because, due to the absence of interference, these thread interleavings differ in only “important ways”. We have shown in previous work how Bogor can

dynamically detect the properties listed above, and how the cost of state-space exploration can be substantially reduced by leveraging this information.

When checking JML specifications with Bogor, it is then quite natural to expose these properties that are desirable for the developer as well as for the model checker engine in the specification language itself. Maybe JML could be extended in some reasonable way to include annotations that allow to describe these useful properties and let them be verified by a tool. As we have seen, Bogor could easily verify these properties. It would be easy to check the method local and thread local properties using a simple marking scheme during the scan of the heap. Finally, read-only declarations are easily checked by maintaining an extra bit of state in Bogor for such objects indicating if they have been initialized, and then one initialization occurs checking that all other accesses are reads.

All the ideas discussed above have the potential of making JML a richer, more mature and more robust specification language. Given the current trend of increase in multi-threaded applications, a language that allowed to express rich properties about both functional and synchronization behavior of concurrent programs, would be more than welcome. The moment seems right to introduce these changes in given that JML is currently going through a very dynamic phase, in which it has opened to changes and enhancements, most notably the work in [19] that is setting the ground for the addition of a universal type system to JML. In fact we are already devoting some efforts, in collaboration with colleagues at University of California at Santa Cruz (Cormac Flanagan) and Iowa State University (Gary Leavens), to extend JML to address some of the issues discussed in this section, specially those related with atomicity.

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, Robby, X. Deng, and J. Hatcliff. Space reductions for model checking quasi-cyclic systems. In *Proceedings of the 3rd International Conference on Embedded Software*, Oct. 2003. (to appear).
5. M. B. Dwyer and V. Wallentine. A framework for parallel adaptive grid simulations. *Concurrency : Practice and Experience*, 9(11):1293–1310, Nov. 1997.
6. 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.
7. R. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Mathematics*, 1967.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Pub. Co., Jan. 1995.
9. S. Hartley. *Concurrent Programming - The Java Programming Language*. Oxford University Press, 1998.
10. J. Hatcliff, Robby, and M. Dwyer. Verifying atomicity specifications for concurrent object oriented software using model checking. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004*, volume 2937 of *Lecture Notes in Computer Science*, pages 175–190. Springer, 2004.
11. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
12. 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.
13. 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.
14. D. Lea. *Concurrent Programming in Java: Second Edition*. Addison-Wesley, 2000.
15. G. T. Leavens, A. L. Baker, and C. Ruby. JML: a Java modeling language. In *Formal Underpinnings of Java Workshop*, Oct. 1998.
16. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. 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.
17. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, and C. Ruby. JML reference manual. Department of Computer Science, Iowa State University. Available from <http://www.jmlspecs.org>, Aug. 2002.
18. B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.
19. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for object structures. Technical Report 424, ETH Zurich, Oct. 2003.
20. 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.
21. A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report 00-03d, Iowa State University, Department of Computer Science, July 2003.
22. M. Richters and M. Gogolla. Validating UML models and OCL constraints. In *The Third International Conference on The Unified Modeling Language (LNCS 1939)*, 2000.
23. 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.
24. Robby, M. B. Dwyer, and J. Hatcliff. Bogor Website. <http://bogor.projects.cis.ksu.edu>, 2003.
25. 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.
26. Robby, E. Rodríguez, M. B. Dwyer, and J. Hatcliff. SpEx Website. <http://spex.projects.cis.ksu.edu>, 2003.
27. D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Jan. 1995.
28. S. D. Stoller. Domain partitioning for open reactive systems. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2002.
29. 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.
30. 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.
31. 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.
32. J. M. Wing. Writing larch interface language specifications. *ACM Trans. Program. Lang. Syst.*, 9(1):1–24, 1987.