JML Runtime Assertion Checking: Improved Error Reporting and Efficiency Using Strong Validity

Patrice Chalin and Frédéric Rioux

Dependable Software Research Group,
Dept. of Computer Science and Software Engineering, Concordia University
chalin@encs.concordia.ca, fred@dsrg.org

Abstract. The Java Modeling Language (JML) recently switched to an assertion semantics based on "strong validity" in which an assertion is taken to be valid precisely when it is defined and true. Elsewhere we have shared our positive experiences with the realization and use of this new semantics in the context of ESC/Java2. In this paper, we describe the challenges faced by and the redesign required for the implementation of the new semantics in the JML Runtime Assertion Checker (RAC) compiler. Not only is the new semantics effective at helping developers identify formerly undetected errors in specifications, we also demonstrate how the realization of the new semantics is more efficient—resulting in more compact instrumented code that runs slightly faster. More importantly, under the new semantics, the JML RAC can now compile sizeable JML annotated Java programs (like ESC/Java2) which it was unable to compile before.

1 Introduction

The assertion semantics of the Java Modeling Language (JML) [13, 17], a behavioral interface specification language for Java, was formerly founded on a classical definition of validity. Elsewhere we have demonstrated that

- this semantics was not faithfully implemented [4] by either of the two main JML tools [1] (namely, `jmlc`, the JML Runtime Assertion Checker (RAC) Compiler and ESC/Java2) and that in any case,
- a comprehensive survey of programmers, mainly from industry, indicated that this is not the semantics that they want [3].

Hence, a new assertion semantics for JML based on "strong validity" was recently proposed [2, 4] and adopted [20, §2.7]. Under such a semantics, an assertion is taken as valid when its evaluation does not result in partial functions being applied to values outside their domain and, the assertion evaluates to true. In terms of runtime assertion checking (RAC), this means that an assertion is considered valid if and only if its evaluation (terminates and) results in true without raising an exception.

We have begun the realization of the new assertion semantics in ESC/Java2 [4]. In this paper, we explain how the JML RAC has been adapted to conform to the new semantics and some of the challenges that we faced. We also demonstrate how the

```
public class Person {
  /*@ spec_public */ private String name; // a spec_public field
  /*@ spec_public */ private int age;      // can be used in public specs
  //@ public invariant age >= 0;
  //@ requires age >= 0;
  public Person(String n, int age) { this.name = n; this.age = age; }

  //@ ensures age == \old(age) + 1;
  public void birthday() { age++; }
}
```

Fig. 1. Person class annotated with JML

realization of the new semantics helped us find new bugs in JML specifications and that it is more efficient, resulting in smaller instrumented bytecode that runs slightly faster. More importantly, under the new semantics, the JML RAC can now compile sizeable JML annotated Java programs (like ESC/Java2) which it was unable to compile before.

This paper first compares both the old classical and new assertion semantics before giving more details on the JML RAC and its design. Then, we present an overview of how we had to modify the JML RAC to support the new assertion semantics and how we assessed the validity of our work.

2 JML Assertion Semantics

Among other things, the Java Modeling Language brings Design by Contract to Java [6, 18]. Hence, in particular, classes can be annotated with invariants and methods adorned with contracts expressed using preconditions and postconditions inside Java comments starting with a leading "@"—see Fig. 1. Invariants, pre- and post-conditions are expressed using *assertions*, which in the case of JML, consist of (the side-effect free subset of) Java `boolean` expressions enhanced with some extra operators and constructs—such as logical implication (==>) and quantifiers (\forall, \exists) [20].

2.1 Classical Assertion Semantics

The old classical JML semantics assumed that assertions, even if their syntax is very close to that of Java, are to be interpreted as formulas of a classical logic. Under such an interpretation, computational issues that can introduce undefinedness such as short-circuiting of logical operators, exceptions, runtime errors, and informal assertions are not explicitly modeled [7, p.29]. Instead, partial functions are modeled as *underspecified total functions* [19]. Hence, partial functions applied to values outside their domains are assigned a fixed, though unspecified value.

2.2 RAC Approximation of the Old Semantics through Game-playing

Conformant with this view of assertions, the JML RAC-compiled bytecode will always consider an assertion as satisfied or violated; it will never be declared as invalid. Since the evaluation of Java expressions can naturally lead to exceptions, the RAC still has to deal with undefinedness. In its attempt to emulate classical two-valued logic from Java's three-valued operational semantics, the RAC resorts to a game-playing strategy as we explain next.

Table 1. Game played by the JML RAC to approximate classical logic

	Value assigned to …		Value of top-level assertion
	`(*informal*)`	`x/0 == y`	
`!(* informal *) && x/0 == y` i.e. *!angelic && demonic*	False	False	False
`! (* informal *) && !(x/0 == y)` i.e. *!angelic && !demonic*	False	True	False
`(* informal *) || !(x/0 == y)` i.e. *angelic || !demonic*	True	True	True

   In the JML RAC, undefinedness comes in two flavors: *demonic* and *angelic* [7, pp.30-31]. Demonic undefinedness arises from various runtime errors or exceptions that are generated when an assertion expression is evaluated. Angelic undefinedness comes from the attempt to evaluate something that is not executable (e.g., an informal predicate or some categories of quantified expression). The JML RAC adopts a game-playing strategy in its attempt to deal with the two kinds of undefinedness. That is, generally, the smallest Boolean subexpression containing an undefined term will be treated as either true or false depending on the evaluation context. For demonic un-definedness the JML RAC tries to choose a truth value for the undefined subexpression that will make the top-level assertion false; whereas for angelic undefinedness the RAC will try to make the top-level assertion true [7]. When both angelic and de-monic undefinedness occurs in the same expression, they each try to influence the top-level assertion in the best way they can to meet their respective goals. Table 1 illustrates the game being played.

   Classical logic does not feature conditional Boolean operators such as conditional conjunction (`&&`). Under the old JML semantics, Java's conditional operators were mapped into their classical non-conditional counterparts. This implied that the JML assertion `E1 && E2` is equivalent to `E2 && E1` [19]. In order to preserve that behavior, the JML RAC evaluated both of its operands when the evaluation of the first operand is exceptional [7, p.27]. Such a scheme can be confusing for developers since it leads to the evaluation of syntactically correct Java expressions differently if done in a Java or JML context as illustrated in Table 2. For both expressions, JML will interpret a logical or between something (possibly) undefined and something true; hence always yielding true in such a case. Java on the other hand will throw an exception upon a null pointer dereference.

2.3   New Semantics Based on Strong Validity

The original JML RAC semantics guessed a truth value for an invalid assertion; using the new semantics, an assertion can be satisfied (evaluated true), violated (false) or

Table 2. Semantic differences between Java and JML

	`true || x.length > 0`	`x.length > 0 || true`
Java	Always `true`	if `x` is not null: `true`. Otherwise: NullPointerException.
JML	Always `true`	Always `true`

invalid (when evaluation does not complete successfully) [4]. Violated and invalid assertions are reported as distinct kinds of error.

Handling Undefinedness.   In our implementation of the new semantics, all logical operators behave in the same way in both Java and JML. For instance, a conditional disjunction or conjunction whose left-hand subexpression is exceptional would cause the resulting expression to be exceptional no matter what the right-hand subexpression refers to. When an exception or runtime error occurs while evaluating part of an assertion, that exception causes the entire assertion to be invalid and the user to be notified. In other words, as soon as demonic undefinedness occurs, the evaluation of the assertion is halted, and the assertion is reported as invalid.

The concept of angelic undefinedness cannot be as easily factored out. As was mentioned earlier, such undefinedness was associated with non-executable subexpressions and was treated in a way that ensured the top-level expression would be "*as true as possible*". We do not want assertion expression evaluation to have the overhead of game playing. In the new semantics, if an assertion is non-executable (in its entirety or in part) then the entire assertion is tagged as non-executable. While most non-executable assertions can be detected at compile-time, the rest can only be discovered at runtime. While it is possible to warn the user that some of the assertions may be non-executable, it is not always possible to precisely say if it will *always* be non-executable [7, 23].

Whether a non-executable assertion should play a role in the overall truth value of a specification depends on what the developer wants. In some cases (e.g. during preliminary development, when there is a higher occurrence of incomplete specifications), one might be willing to ignore them by treating the assertion in which they occur as equivalent to true. However, in other situations, to gain extra confidence and ensure that the specifications are entirely verified, one may prefer to have non-executable assertions be reported and make the specification verification fail since they cannot be enforced or verified. Non-executable assertions can either be simplified to true or false, depending on the setting of a JML RAC compilation flag. Trying to factor out non-executable expressions from an assertion while trying to infer a truth value to the expression would mimic the game played by the previous semantics. We believe non-executable assertions should be avoided as they provided very little in the context of automated program verification.

3   JML Runtime Assertion Checker (RAC), Old Semantics

The JML RAC is part of the Common JML tools suite—formerly known as the ISU JML tool suite. It uses a "compilation-based approach" for translating JML specifications into runtime checking bytecode [7]. Unlike static checkers, which verify program properties at compile-time, the JML RAC enables dynamic checking by generating bytecode that verifies that specifications are satisfied during program execution. When an assertion fails, the JML RAC-compiled code generates a runtime error. The remainder of this section describes the design and operation of the RAC under the old semantics. (As will be seen in the next section, the design changes required to implement the new semantics have been quite localized.)

The JML RAC is built on top of the MultiJava (MJ) compiler and uses the JML Checker[1] for type checking JML specifications and as the front-end for building an Abstract Syntax Tree (AST). JML specification clauses are translated

```
public int x, y;

//@ requires b && x < y;
public void m(boolean b) {...}
```

Fig. 2. A simple method precondition

into *assertion methods*. For each Java method, three RAC assertion methods are generated: one for precondition checking and two for postcondition checking (i.e., for normal and exceptional termination). The JML-specified Java methods are instrumented using a wrapper approach. The instrumentation process takes the original body of a method and extracts it into a private method with a uniquely defined name. The original signature of the method is used for the newly created wrapper method hence replacing it. The wrapper implements the specification checking logic by calling the original body and assertion methods when required. Not only are the preconditions and postconditions associated with the method called, but some class-related assertion methods are also called (e.g., for invariant and constraint checking [18]). The control flow of the wrapper approach to method instrumentation is presented in [7, §4.1].

3.1   Code Instrumentation

Every Java class compiled with the JML RAC contains not only its normal content (as would be generated by, e.g., `javac`), but also an embedding of its specification and how to verify it at runtime. Instrumentation code is generated on a per classifier, per method, per field, and per assertion basis [9]. Most generated instrumentation code gives rise to an overhead that is linear and foreseeable, though for assertion expressions interpreted under the old semantics it used to be polynomial (at least quadratic!) as we shall soon illustrate.

3.1.1   General Assertion Evaluation

Under the old semantics, JML RAC-generated code that evaluates an assertion expression tended to be rather verbose because expression evaluation had to emulate classical two-valued logic while playing an optimization game with angelic and demonic undefinedness. Hence, for example, the JML RAC made extensive use of new variables: as a rule of thumb, *every* subexpression had an associated new internal variable used to hold its value. Moreover, each step in the evaluation was done separately and had again its own new internal variable, and sometimes its own try block. For example, a simple precondition such as the one given in Fig. 2 was translated into 59 lines of instrumentation code and used 7 new internal variables—see Fig. 3. Upon reading the code, one may notice the right-hand side of the `&&` operator is evaluated if the left-hand side is exceptional; as mentioned earlier, this different from the Java semantics for that operator.

3.1.2   RAC Generated Code Exceeded JVM and Class File Format Limits

The JML RAC's attempt to implement an assertion semantics based on classical two-valued logic caused the instrumented code to be *much* larger than the source. We note

---

[1] MultiJava: http://multijava.sourceforge.net/; Java Modeling Language (JML): http://www.jmlspecs.org/.

```
 1| try {                                              31|              try {
 2|   // eval of &&                                     32|                  rac$v6 = this.y;
 3|   boolean rac$v0 = true;                            33|              }
 4|   boolean rac$v1 = false, rac$v2 = false;           34|              catch (JMLNonExecutableException jml$e0) {
 5|   // arg 1 of &&                                     35|                  rac$v4 = true;
 6|   try {                                              36|              }
 7|     rac$v0 = b;                                      37|              catch (java.lang.Exception jml$e0) {
 8|   }                                                  38|                  rac$v3 = true;
 9|   catch (JMLNonExecutableException jml$e0) {         39|              }
10|     rac$v2 = true;                                   40|          }
11|   }                                                  41|          if (rac$v3) { rac$v0 = false; }
12|   catch (java.lang.Exception jml$e0) {               42|          else if (rac$v4) { rac$v0 = true; }
13|     rac$v1 = true;                                   43|          else try {
14|   }                                                  44|              rac$v0 = rac$v5<rac$v6;
15|   if (rac$v0) {                                      45|          }
16|     // arg 2 of &&                                   46|          catch (JMLNonExecutableException jml$e0) {
17|     try {                                            47|                  rac$v0 = true;
18|       boolean rac$v3 = false, rac$v4 = false;        48|          }
19|       int rac$v5 = 0;                                49|          catch (java.lang.Exception jml$e0) {
20|       int rac$v6 = 0;                                50|                  rac$v0 = false;
21|       try {                                          51|          }
22|               rac$v5 = this.x;                       52|      }
23|       }                                              53|      catch (JMLNonExecutableException jml$e0) {
24|       catch (JMLNonExecutableException jml$e0) {     54|          rac$v2 = true;
25|               rac$v4 = true;                         55|      }
26|       }                                              56|      catch (java.lang.Exception jml$e0) {
27|       catch (java.lang.Exception jml$e0) {           57|          rac$v1 = true;
28|               rac$v3 = true;                         58|      }
29|       }                                              59| }
30|       if (!rac$v3) {                                 60| }
```

Fig. 3. Instrumentation code evaluating "requires b && x < y" (old RAC semantics)

here that in some cases, the generated code was so large that a Java compiler would be unable to process it. For example, in the ESC/Java2 project, there are a few classes that could not be compiled using the JML RAC. One of these classes (`javafe.ast.-TypeDeclElem`) has an automatically generated postcondition composed of a conjunction of 118 implications. Unfortunately, the instrumented code generated for verifying the postcondition consisted of 15,816 lines of Java which no compiler can successfully compile. This is because the assertion method which checks the postcondition had a top-level catch block that was too far away from its try block (due to limitations of the JVM instruction set and the Java class file format, the two blocks must compile into byte code that is no more than 65535 bytes apart [21, §4.10]). Of course, methods with such postconditions are rather rare, but the fact is that the evaluation of expressions following the original semantics JML RAC does not scale and cannot cope with heavily specified code. Users should obtain benefits from writing richer specifications rather than be penalized.

4   JML RAC Redesign in Support of Strong Validity

4.1   Expression Evaluation

Representing Assertions as a Single Java Expression. Most of the time, an assertion's body can be evaluated exactly as written (i.e., without having to declare new variables for each subexpression). The possible outcomes of such an evaluation are true, false, or an error/exception. Hence, if we choose not to model partial functions by

underspecified total functions, the evaluation of expressions becomes quite straightforward, helping us overcome some of the practical limitations mentioned earlier.

Expression Evaluation under the Old Semantics. Expression-evaluation code was generated by RAC expression translators implemented as AST visitors [12]. Given an expression to be translated, an expression translator would walk the expression AST and build a compound "node" that contained the instrumentation code to evaluate the expression at runtime. Such a node could either be wrapped with a try-catch block or not. A high-level translator used such nodes whenever it needed for that expression to be evaluated while generating the wrapper assertion methods [7].

The translation process was achieved by visiting every subexpression of a given top-level expression and generating nodes to evaluate the subexpressions. As was mentioned earlier, a new variable was usually defined to hold the value of a subexpression. Each of the subexpression nodes was stored on a stack. When all of the expressions had been visited, the stack of nodes was used to generate an all encompassing node that, most often, was wrapped in a try-catch block before being returned to the sender. We briefly note that proper handling of quantified expressions is quite involved (e.g. requiring specialize heuristic-based static analysis in order to decide how, if possible, to evaluate them). Hence we will only describe the handling of quantified expressions in Section 0 for the new semantics.

New Approach to Expression Evaluation. The implementation of the new semantics requires alternate expression translators. For this reason, we created a new general expression translator usable almost as a simple drop-in replacement for the old (top-level) translators. Under the new semantics, there is no need to evaluate subexpressions separately through the use of newly declared variables. Precedence of operators is embedded in the AST and hence, an appropriate use of parentheses while visiting the tree avoids the need for variables. Since, in the new semantics, a clause is either entirely executable or not at all, a new runtime exception was created to short-circuit evaluation code generation in the event that one of the subexpressions is found to be non-executable at compile-time. At runtime, the expression is evaluated in a top-level try-block that catches two things: (i) `JMLNonExecutableException`s (Fig. 4), and (ii) all other errors and exceptions. `JMLNonExecutableException`s cause an entire assertion to immediately simplify to true (as was mentioned in Section 0, a command line option allows developers to change this default to false). Any other exception or error thrown while evaluating an expression is caught and wrapped in a `JMLEvaluationError` before being re-thrown.

4.2 Handling Quantified Expressions

Quantified expressions, unlike other simpler expressions, cannot have their evaluation code mechanically derived. They have to be analyzed beforehand. In order to properly analyze quantified expressions and derive the best way to verify them at runtime, the JML RAC provides a special package and translator that, like the other high-level translators, uses the (base) expression translator to evaluate expressions. In order to reuse the existing quantifier evaluation package while implementing our new direct expression evaluation approach, we decided to wrap the output of the quantifier translator into an inner class that is used in the evaluation of the assertion instead of the quantified expression as described in [23].
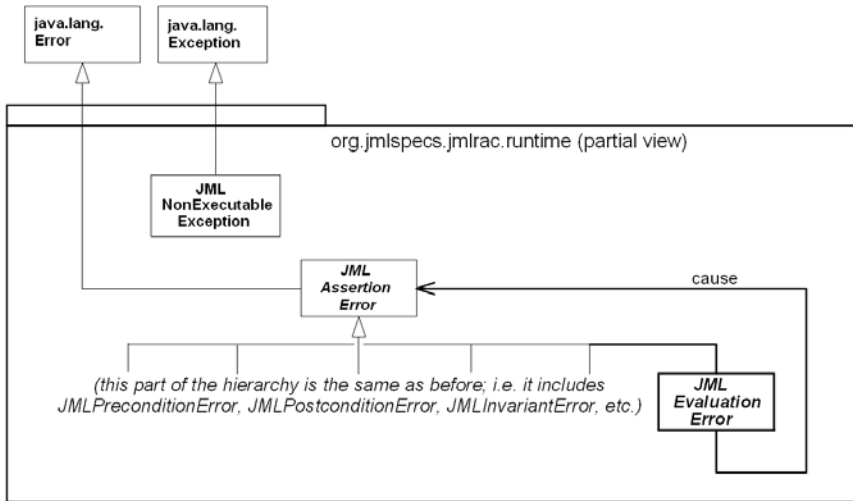
Fig. 4. Modified error hierarchy of the JML RAC runtime package

```java
public static ArrayList myList = new ArrayList();

//@ requires (x > y) || (x < 0) ||
//@   (\forall Object obj;
//@        myList.contains(obj); obj instanceof Integer);
public void m(int x, int y) {...}
```

Fig. 5. Sample assertion containing a quantified subexpression

Consider the specification fragment of Fig. 5. Under the new semantics, RAC code for the method's precondition is as shown in Fig. 6. Note that the try block starts with the definition of the inner class `rac$v4` whose `eval()` method performs the evaluation of the quantified subexpression. The statement following the inner class definition instantiates the class. Finally, the last statement of the try block, marked (*), is the one that can be clearly seen to correspond to the requires clause expression of Fig. 5.

5   Validation: Assessment and Statistics

Basic Validation: Regression Testing. The Common JML tool suite is supported by an extensive collection of automated tests. These tests, numbering in the thousands, help developers ensure the integrity of the tool suite following any modification. The test suite for the JML RAC consists of approximately 500 test files, each containing several test cases. Out of those, more than 375 are grouped under the `racrun` package, whose purpose is to test the runtime behavior of RAC-compiled code—this is in contrast to, e.g., testing the behavior of the JML RAC. In particular, the `racrun` package is meant to test all JML statements and expressions individually and in various combinations. The test coverage of the `racrun` package is considered sufficiently complete. For the purpose of testing the new assertion semantics we adapted the `racrun`

```
try {
    class rac$v4 {
        public boolean eval() {
            boolean rac$v0 = false;
            java.util.Collection rac$v1 = new java.util.HashSet();
            java.util.Collection rac$v3 = MyClass.myList;
            rac$v1.addAll(rac$v3);
            java.util.Iterator rac$v2 = rac$v1.iterator();
            rac$v0 = true;
            while (rac$v0 && rac$v2.hasNext()) {
                java.lang.Object obj = (java.lang.Object) rac$v2.next();
                rac$v0 = (!(MyClass.myList.contains(obj))
                            || obj instanceof java.lang.Integer);
            }
            return rac$v0;
        }
    }
    rac$v4 rac$v0Evaluator = new rac$v4();
    rac$pre4 = (((x > y) || (x < 0)) || rac$v0Evaluator.eval()); // (*)
} catch (JMLNonExecutableException rac$v5$nonExec) {
    rac$pre4 = true;
} catch (Throwable rac$v6$cause) {
    JMLChecker.exit();
    throw new JMLRacExpressionEvaluationError("Invalid Expression in
        \"FM08.java\", line 36, character 10", rac$v6$cause);
}
```

Fig. 6. RAC Code for precondition evaluation of method m()

package to support the expected output of the new semantics and ensured that all unit tests passed successfully.

Testing Code Generation Robustness. Aside from `racrun` tests, we also successfully compiled all the JML *model classes*, which are heavily annotated classes that specify abstract data types such as sequences, sets and bags. The model classes extensively use of the features of JML. Such a test suite helped us discover some design flaws that surfaced in rare circumstances. Most of them were for situations where operator precedence is not preserved during the translation from JML to Java.

While the `racrun` package gave us confidence in the behavior of the generated code, ensuring that the model classes could yield properly formed instrumented source code when compiled using the new assertion semantics demonstrated the robustness of the code generation for the new semantics.

Assessing Improved Capabilities. One of the goals of the JML community is to use its own tools. As was mentioned earlier, prior attempts to compile ESC/Java2 [11] with the JML RAC demonstrated that for a few source files, it would generate instrumentation code that had such large try blocks that it was impossible to represent them

```
1|try {
2|   rac$pre0 = (b && (this.x < this.y));
3|} catch (JMLNonExecutableException rac$v0$nonExec) {
4|   rac$pre0 = true;
5|} catch (Throwable rac$v1$cause) {
6|   JMLChecker.exit();
7|   throw new JMLRacExpressionEvaluationError("Invalid Expression in \"...\", line 5, ...", rac$v1$cause);
8|}
```

Fig. 7. Evaluation of precondition in the modified RAC

Table 3. ESC/Java2 source code statistics for `escjava` and `javafe` packages

Source code size (measured in MB)	Old Semantics	New Sem.	Δ	New/Old
`Escjava` Instrumented source code	33.6	26.5	7.1	≈ 78.9%
`Escjava` Instrumented bytecode	12.2	9.8	2.4	≈ 80.3%
`Javafe` Instrumented source code	35.5 *30.5*[*]	21.7 *21.6*[*]	13.8 *8.9*[*]	≈ 61.1% ≈ *70.8%*[*]
`Javafe` Instrumented bytecode	10.7	8.0	2.7	≈ 74.8%

[*] Adjusted measurement in which we removed the code size for two files that could not be compiled under the old semantics due to the excessive size of try blocks in the instrumentation code.

in bytecode. We verified that with the new semantics, such a problem did not happen as all files were amenable to RAC compilation. Moreover, for the files that compiled using both semantics, we gathered statistics to measure the overall reduction in code size.

Measurements and Code Size Statistics. Throughout our assessment of the new semantics, we gathered some measurements that demonstrated an improvement in both size and performance of JML RAC-instrumented code. In order to understand the source of such an improvement, one should consider that the new semantics generates much less code to evaluate expressions than the previous semantics did, in part due to its more coarse approach, but mostly because it no longer plays the angelic vs. demonic undefinedness game for invalid expressions. Moreover, that code always takes advantage of short-circuited logical operators and does not try to assign a truth value to exceptional expressions. For instance, the instrumentation code of Fig. 3 (59 LOC) is reduced to only 8 lines of code under the new semantics—Fig. 7.

We observed that the `racrun` test package executes on average 8% faster than the version using the original semantics (average of 96.0s vs. 88.3s in five independent runs). Such an evaluation includes the parsing, checking, code generation, compilation, run and validation against expected output files of over 375 tests files. While using the new semantics on ESC/Java2 source, both the generated instrumented source code and bytecode showed a significant size reduction, as illustrated in Table 3. E.g., for the ESC/Java2 `escjava` package (301 classes), the instrumented source code using the new semantics was only 78.9% the size of the one instrumented with the original semantics (for this metric, we had the JML compiler emit the Java source corresponding to the instrumented runtime checking code that it would otherwise create a `.class` file for). For the instrumented bytecode, the new/original semantics ratio was of 80.5%. The compiler front end of ESC/Java2 (`javafe` package, 216 classes) displayed similar improvements in size. Two classes in the `javafe` package could not be compiled under the old semantics due to the excessive size of try blocks. If we factor out those two abnormally large (and uncompilable) instrumented source files, the new/original size ratio goes from 61.1% to 70.8%.

6   Effectiveness at Finding Bugs

Using the new RAC semantics, approximately 45 previously undetected errors have been found in JML specifications of the sample and model files (307 files, 77KLOC) included with the JML distribution. It should be noted that most of these sample and model files have been in use since 2002; some are from published specifications that

```
public /*@pure*/ class NaturalNumber implements TotallyOrderedCompareTo,... {
  /*@ spec_public */ private final BigInteger value;

  //@ public normal_behavior
  //@   ensures \result == value.compareTo(n.value);
  public int compareTo(NaturalNumber n) {
      return value.compareTo(n.value);
  }

  //@ also public normal_behavior
  //@  requires o instanceof NaturalNumber ...
  //@  ensures \result == ...;
  public int compareTo(Object o) throws ClassCastException {
      return value.compareTo(((NaturalNumber)o).value);
  }

  /*@ public normal_behavior
    @   requires !isZero() && exponent.equals(ZERO);
    @   ensures \result.equals(ONE);
    @ also
    @   forall NaturalNumber v;
    @   requires !(exponent.equals(ZERO)) &&
    @      exponent.compareTo(BigInteger.valueOf(Integer.MAX_VALUE)) <= 0; // (*)
    @ ... */
  public NaturalNumber pow(NaturalNumber exponent) { ... }
...
```

Fig. 8. Excerpt from the JML model class **NaturalNumber**

have appeared in peer-reviewed books or articles and hence have been carefully been reviewed by both human readers and/or analyzed by other JML tools. Of the errors found, slightly more than half are the specification equivalent of common programming errors (such as null dereferences[2] and array index out of bound errors) as well as some common object-oriented programming pitfalls.

As an example of the latter, consider the excerpt of the `NaturalNumber` model class, slightly simplified due to space constraints, given in Fig. 8. (Note that all declarations of reference types, with the exception of local variables, are non-null by default in JML unless annotated with `/*@nullable*/` [5].) Under the new semantics, unit testing reports a `ClassCastException` during the precondition evaluation of the second spec-case at (*). Given this information one can easily see that the developer forgot to include a `compareTo(BigInteger)` method, and hence the call to `compareTo` at (*) resolves to `compareTo(Object)`, resulting in a meaningless method contract due to the undefinedness of the precondition.

Most of the remaining errors had to do with recursive specification constructs. A simple example of such an error is illustrated by the excerpt from the specification for `java.lang.Boolean` given in Fig. 9. Notice how the model field `theBoolean` is represented by the expression "`booleanValue()`" and yet, the contract of `booleanValue()` defines it to be equal to the model field `theBoolean`. Hence each is defined in terms of the other. (Note that a JML *model field* is a specification-only field used to represent an abstraction of part of an object's or a class' state, for non-static and static fields respectively. The binding between the model field's value and the concrete state is given in the form of a `represents` clause.)

---

[2] A potential null-dereference is shown in Fig. 10—see the underlined occurrence of `nextNode`.

```
public final /*@ pure @*/ class Boolean ... {
  //@ public model boolean theBoolean;
  //@ represents theBoolean <- booleanValue();

  /*@ public normal_behavior
    @   assignable \nothing;
    @   ensures \result == theBoolean;
    @*/
  public boolean booleanValue();
  …
}
```

Fig. 9. Excerpt from the JML API specification for **java.lang.Boolean**

```
//@ model import org.jmlspecs.models.JMLObjectSequence;

public class OneWayNode {  // Singly Linked Node
  /*@spec_public*/ protected /*@nullable*/ Object entry;
  /*@spec_public*/ protected /*@nullable*/ OneWayNode nextNode;

  //@ public model JMLObjectSequence entries;
  //@ public model JMLObjectSequence allButFirst; ...
  //@ protected represents entries <- allButFirst.insertFront(entry);
  //@ protected represents allButFirst <- (nextNode == null)
  //@   ? new JMLObjectSequence() : nextNode.entries;
  ...
}

public class TwoWayNode extends OneWayNode { // Doubly Linked Node
  /*@spec_public*/ protected /*@nullable*/ TwoWayNode prevNode;

  //@ public model JMLObjectSequence prevEntries; ...
  //@ protected represents prevEntries <- (prevNode == null)
  //@   ? new JMLObjectSequence()
  //@     : prevNode.prevEntries.insertBack(prevNode.entry);

  /*@ public normal_behavior
    @  assignable prevEntries;
    @  ensures prevEntries.equals(\old(prevEntries).insertBack(newEntry))
    @        && \not_modified(nextNode.entries);
    @*/
  public void insertBefore(/*@nullable*/ Object newEntry) {...}
}
```

Fig. 10. Classes from the **org.jmlspecs.samples.list.node** package

An example that is more involved is treated next. Consider the code excerpt from two classes in `org.jmlspecs.samples.list.node` package (Fig. 10): `OneWayNode`, used to build singly-linked lists and `TwoWayNode` used to build doubly-linked lists[3]. A `OneWayNode` contains an entry and a possibly-null reference to a next node (`nextNode`). `TwoWayNode`s extend `OneWayNode`s by adding a possibly-null reference to a previous node (`prevNode`). Two model fields are defined for a `OneWayNode`: `entries` which is the sequence of `Object` entries contained in the linked list rooted at `this` node; `allButFirst`, as the name implies, is the sequence of `Object` entries contained in the linked list rooted at `this.nextNode`, provided it is not null. Similarly, `TwoWayNode` defines the model field `prevEntries` to be the sequence of entries contained in the linked list rooted at this node but by following the `prevNode` field.

---

[3] Note that in the JML distribution, the specifications for these two classes are each spread over three files since use was made of JML's specification refinement feature. Since this feature is somewhat involved, a simplified—though equivalent—version of the classes and their specifications is given here.

Initially `curr == this.prevNode`.
Illustrated is: `this.prevEntries (curr) = <b, c, ... >`.
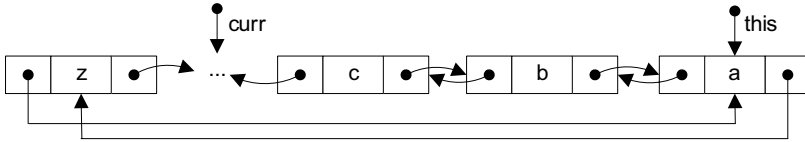Terminates when `curr == null` or `curr == this`.

Fig. 11. Evaluation of **TwoWayNode.prevEntries** (terminates even for circular lists)

Running the test suite for this package using RAC instrumented versions of these classes reports a null-dereference error on `nextNode.entries` in the postcondition of `TwoWayNode.insertBefore()`. More interestingly is the fact that under the new semantics, a stack overflow error is reported. While at first we believed that the error might have been caused by a bug in our implementation of the new assertion semantics, inspection of the error reports allowed us to identify bugs in the specifications. Notice that the definitions of the representations of `OneWayNode.entries`, `OneWayNode.allButFirst` and `TwoWayNode.prevEntries` are all subject to looping forever if the nodes are part of a cyclic list. The main point here is that under the old semantics, the stack overflow caused by the use of any one of these three model fields would have been caught and translated into some truth value that would make true the overall assertion in which they occurred as subexpressions! A corrected specification for `prevEntries` is given in Fig. 12—the corrections for the other two model fields are

```
public class TwoWayNode extends OneWayNode
{
  protected /*@nullable*/ TwoWayNode prevNode;
  /*@ public model JMLObjectSequence prevEntries; ...
    @ protected represents prevEntries <- prevEntries();
    @
    @ public model pure JMLObjectSequence prevEntries() {
    @    // To detect cycles we use a helper function.
    @   return prevEntries(prevNode);
    @ }
    @
    @ public model pure
    @   JMLObjectSequence prevEntries(nullable TwoWayNode curr) {
    @   return (curr == null
    @            // the following disjunct prevents infinite recursion
    @            || curr == this)
    @          ? new JMLObjectSequence()
    @          : prevEntries(curr.getPrevNode()).insertBack(curr.getEntry());
    @ }*/

  ...

  /*@ public normal_behavior
    @    assignable prevEntries;
    @    ensures prevEntries.equals(\old(prevEntries).insertBack(newEntry))
    @       && (nextNode != null ==> \not_modified(nextNode.entries));
    @*/
  public void insertBefore(/*@nullable*/ Object newEntry) {
    ...
  }
}
```

Fig. 12. TwoWayNode **specification now correctly handling lists with cycles**

similar. Note how `prevEntries` is represented by the `prevEntries()` model method which returns the sequence of entries in the nodes reachable from `this` by following `prevNode` links until either `null` is reached or we have cycled by to `this`—see the illustration of **Fig.** 11. With this change, all tests pass under the new semantics.

7 Related Work

The use of program assertions as an aid in verifying the correctness of programs was first explored by Alan Turing in the late 40s. Over the decades, this idea was further refined by Computer Science founding fathers such as Goldstine, von Neumann, McCarthy, Floyd, Hoare and others [15, 16]. An early milestone in this vein was Hoare's 1969 Axiomatic Basis for Computer Programming where the pre- and post-conditions of Hoare triples were expressed by means of assertions [14]. Early on, assertions were also introduced as a distinct construct in mainstream programming languages, and eventually, Hoare triples found their way into the programmer's tool box in the form of a method known as Design by Contract (DbC) [22]. A comprehensive report on the history of runtime assertion checking can be found in Clarke and Rosenblum's IMPACT report [10].

To our knowledge, all programming languages (or programming language extensions) supporting the use of plain inline assertions or DbC at runtime also support an assertion semantics like the one recently adopted for JML. The main reason is that it results in the simplest and most efficient runtime checking code. It remains a challenge to find a proper balance between minimalist instrumentation code with less useful error reporting in cases where assertions fail due to undefinedness vs. more accurate error reporting (which requires extra try-catch blocks to catch exceptions and wrap them up in another more meaningful exception before re-throwing it).

Another related challenge is to preserve soundness of the new assertion semantics in the context of static checkers like ESC/Java2. In [3, 4], we show how this can be achieved by making use of definedness predicates. Use of definedness predicates allows us to keep on using provers for classical logic (even though the new assertion semantics is essentially that of a three-valued logic). We note that the increase in processing time required for definedness checking in ESC/Java2 is currently less than 2% [4]. This is fairly small compared to the increase in static error detection offered by the adoption of strong validity (especially for API specifications for which little more than type checking was provided before).

A few other tools that make direct or indirect use of `jmlc` will be able to upgrade to the new semantics merely by making use of the new version of the compiler. This is the case for JmlUnit, a tool that can help developers create JUnit tests using JML specifications as oracles [8], and the `SpEx`-JML model checker [26]. Being build on the Bogor framework [24], `SpEx`-JML makes use of `jmlc`'s core translation module to render runtime checking code for JML constructs. By making use of the new version of `jmlc`, `SpEx`-JML will in effect be implementing a form of model checking based on three-valued logic (in the spirit of [25]).

8   Conclusion

The work reported here was conducted as part of an ongoing effort to bring strong validity into all of the main JML verification tools. Using ESC/Java2, we have demonstrated the effectiveness of the new semantics by showing how it uncovered about 50 errors in the (143) API specifications of the `java.*` package [4, §6.1]. In applying the JML RAC (also with the new semantics), we have uncovered a comparable number of bugs in the JML model classes and specification samples which are a part of the JML distribution.

As future work we plan to finalize a few unresolved issues. For example, under the former RAC semantics, the value of `\old` expressions[4] that occur in method postconditions are evaluated before preconditions. This often leads to runtime errors under the new semantics since the evaluation of the `\old` expressions is meant to be guarded by the preconditions. That is, the pre-state evaluation of the `\old` expressions should be done if and only if the corresponding preconditions evaluate to true. We also need to better address the issue of short circuiting the evaluation of an assertion when one of its subexpressions raises an exception because it is non-executable. Currently we simplify the entire assertion to true; ideally we would want to simplify the smallest top-level conjunct that contains the non-executable subexpression.

References

[1] Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An Overview of JML Tools and Applications. International Journal on Soft-ware Tools for Technology Transfer (STTT) 7(3), 212–232 (2005)

[2] Chalin, P.: Reassessing JML's Logical Foundation. In: Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP 2005), Glasgow, Scotland (July 2005)

[3] Chalin, P.: Are the Logical Foundations of Verifying Compiler Prototypes Matching User Ex-pectations? Formal Aspects of Computing 19(2), 139–158 (2007)

[4] Chalin, P.: A Sound Assertion Semantics for the Dependable Systems Evolution Verifying Compiler. In: Proceedings of the Int'l Conf. on Soft. Eng. (ICSE), pp. 23–33 (2007)

[5] Chalin, P., James, P.R.: Non-null References by Default in Java: Alleviating the Nullity Annotation Burden. In: Proceedings of the ECOOP, pp. 227–247 (2007)

[6] Chalin, P., Kiniry, J., Leavens, G.T., Poll, E.: Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006)

[7] Cheon, Y.: A Runtime Assertion Checker for the Java Modeling Language., Iowa State Uni-versity, Ph.D. Thesis, also TR #03-09 (April 2003)

[8] Cheon, Y., Leavens, G.T.: A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In: Proceedings of the ECOOP, pp. 231–255. Springer, Heidelberg (2002)

[9] Cheon, Y., Leavens, G.T.: A Contextual Interpretation Of Undefinedness For Runtime Assertion Checking. In: Proc. Int'l Symp. on Automated Analysis-driven Debugging (2005)

---

[4] The occurrence of \old(*e*) in a post condition refers to the pre-state value of *e*.

[10] Clarke, L.A., Rosenblum, D.S.: A Historical Perspective on Runtime Assertion Checking in Software Development. ACM SIGSOFT SEN 31(3), 25–37 (2006)

[11] Cok, D.R., Kiniry, J.R.: ESC/Java2: Uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)

[12] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Ob-ject-Oriented Software. Addison-Wesley, Reading (1995)

[13] Gary, T.L., Albert, L.B., Clyde, R.: Preliminary Design of JML: A Behavioral Interface Specification Language for Java. SIGSOFT Softw. Eng. Notes 31(3), 1–38 (2006)

[14] Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)

[15] Hoare, C.A.R.: Assertions: A Personal Perspective. IEEE Annals of the History of Comput-ing 25(2), 14–25 (2003)

[16] Jones, C.B.: The early search for tractable ways of reasoning about programs. IEEE Annals of the History of Computing 25(2), 26–49 (2003)

[17] Leavens, G.T., Baker, A.L., Ruby, C.: JML: A Notation for Detailed Design. In: Haim Kilov, B.R., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems, pp. 175–188. Kluwer, Dordrecht (1999)

[18] Leavens, G.T., Cheon, Y.: Design by Contract with JML (2006), http://www.jmlspecs.org

[19] Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML ac-commodates both runtime assertion checking and formal verification. Science of Com-puter Programming 55(1-3), 185–208 (2005)

[20] Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual (2007), http://www.jmlspecs.org

[21] Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Prentice-Hall, Engle-wood Cliffs (1999)

[22] Meyer, B.: Applying Design by Contract. Computer 25(10), 40–51 (1992)

[23] Rioux, F.: Effective and Efficient Design by Contract for Java. M.Comp.Sc. thesis, Con-cordia University, Montréal, Québec (2006)

[24] Robby, E., Rodríguez, M.B., Dwyer, M.B., Hatcliff, J.: Checking JML specifications us-ing an extensible software model checking framework. International Journal on Software Tools for Technology Transfer (STTT) 8(3), 280–299 (2006)

[25] Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM To-PLaS 24(3), 217–298 (2002)

[26] SAnToS, SpEx Website (2003), http://spex.projects.cis.ksu.edu