Effective and Efficient Runtime Assertion Checking for JML Through Strong Validity

Frederic Rioux and Patrice Chalin

Dependable Software Research Group,
Dept. of Computer Science and Software Engineering,
Concordia University
www.dsrg.org

Abstract. Previously, we presented an assertion semantics for JML 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), a behavioral interface specification language for Java, is founded on a classical definition of validity. Elsewhere we have demonstrated that

- this official JML semantics is not faithfully implemented by either of the two main JML tools—namely the JML Runtime Assertion Checker (RAC)[1] and ESC/Java2 [Chalin07b]—and that in any case,
- a comprehensive survey of programmers, mainly from industry, indicated that this is not the semantics that they want [Chalin07a]

Hence, a new assertion semantics based on “strong validity” was recently proposed for JML [Chalin05,Chalin07b]. 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 it evaluates to true without raising an exception.

Work has begun on the realization of the new assertion semantics in ESC/Java2 [Chalin07b]. 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 realization of the new semantics 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 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.

[1] Also known as the JML compiler, jmlc.

2 JML Assertion Semantics

2.1 Classical Assertion Semantics

The classical JML semantics assumes that assertions, even if their syntax is very close to that of Java, are interpreted as formula of a classical logic. Under such an interpretation, computational issues that can introduce undefinedness such as short-circuit of logical operators, exceptions, runtime errors, and informal assertions are not explicitly modeled [Cheon03, p.29]. Instead, partial functions are modeled as *underspecified total functions* [Leavens+05]. Hence, partial functions applied to values outside their domains are assigned a fixed, though unspecified value.

2.2 RAC Approximation JML 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.

In the JML RAC, undefinedness comes in two flavors: *demonic* and *angelic* [Cheon03, 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 undefinedness 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 [Cheon03, pp.30-31]. When both angelic and demonic 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 (`&&`). JML maps Java's conditional operators into their classical non-conditional counterparts. This implies that the JML assertion `E1 && E2` is equivalent to `E2 && E1` [Leavens+05]. In order to preserve that behavior, the JML RAC evaluates both of its operands when the evaluation of the first operand is exceptional [Cheon03, 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

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

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

Table 2: Semantic differences between Java and JML

exception upon a null pointer dereference.

2.3 New Semantics Based on Strong Validity

The original JML RAC semantics guesses a truth value for an invalid assertion; using the new semantics, an assertion can be satisfied (evaluated true), violated (false) or invalid (when evaluation does not complete successfully) [Chalin07b]. Violated and invalid assertions are reported by distinct kinds of error.

2.3.1 Handling Undefinedness

In our new implementation, all logical operators behave 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 mentioned earlier, such undefinedness was associated with non-executable subexpression and was treated in a way that ensures 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 our alternative semantics, if an assertion is non-executable (in its entirety or in part) then the entire assertion is tagged as non-executable. Most of the non-executable assertions can be detected at compile-time (e.g., assertions with informal predicates or unbound quantifiers), but a few of them can only be discovered at runtime (e.g. when a JML model field is used in the specification, but the abstract model field is not given a concrete representation). 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.

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. For that reason, non-executable assertions can either fold to true or false, depending on the setting of a JML RAC compilation flag.

3 JML Runtime Assertion Checker (RAC)

3.1 Overview

The JML RAC is part of the Common JML tools suite[2]. It uses a "compilation-based approach" for translating JML specifications into runtime checking bytecode [Cheon03, pp.10-11]. Unlike static checkers which verify program properties at compile-time, the JML RAC enables

[2] Formerly known as the ISU JML tool suite.

dynamic checking by generating bytecode that verifies that specifications are satisfied during program execution. Whenever one of the assertions fails, the JML RAC-compiled code generates a runtime error.

3.2 Design

The JML RAC is built on top of the MultiJava[3] (MJ) compiler and uses the JML Checker[4] for type checking JML specifications and as the front-end for building an Abstract Syntax Tree (AST). JML specification clauses are translated 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 [Cheon03, pp. 47-53]. The instrumentation process takes the original body of a method and extracts it out into a private method with a uniquely defined new name. The original signature of the method is used for the newly created wrapper method that is to replace it. The wrapper implements the specification checking logic and calls 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 [Leavens+06]). The control flow of the wrapper approach to method instrumentation is presented in [Cheon03 p.49, Dai05 p.24].

3.3 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 is generated on a per classifier, per method, per field, and per assertion basis [Cheon+05]. While most instrumentation generates an overhead that is linear and foreseeable, yet for assertion expressions it is polynomial (at least quadratic!) as we shall soon illustrate.

3.3.1 General Assertion Evaluation

JML RAC-generated code that evaluates an assertion expression tends to be rather verbose. Expression evaluation is considerably more involved as it is tightly related to the original RAC's semantics. The JML RAC makes extensive use of new variables. As a rule of thumb, *every* subexpression has an associated new internal variable. Moreover, each step in the evaluation is done separately and has again its own new internal variable, and sometimes its own try block. For example, a simple precondition such as the one given in Figure 1 is translated into 59 lines of instrumentation code and uses 7 new internal variables. The resulting generated code is presented in Figure 2. 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] The MultiJava Project, http://multijava.sourceforge.net/
[4] The Java Modeling Language (JML), http://www.jmlspecs.org/

```
public int x, y;

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

Figure 1: Method with a simple precondition

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

Figure 2: Instrumentation code to evaluate "requires b && x < y" in the RAC (classical semantics)

3.3.2 Some RAC Generated Code Cannot Be Compiled by a Java Compiler

The JML RAC's attempt to implement an assertion semantics based on classical two-valued logic causes the instrumented code to be *much* larger than the source. We note here that in some cases, the generated code is so large that a Java compiler is unable to process it. For example, in the ESC/Java2[5] project, there are a few classes that cannot be compiled by 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 consists of 15,816 lines of Java which no compiler can successfully compile.

The reason that no compiler can process this code is that the assertion method which checks the postcondition has a top-level catch block that is 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 [Lindholm+99, §4.10]). Of course, methods with such postconditions are rather rare, but the fact is that the evaluation of expressions as implemented in the original JML RAC does not scale and cannot compile heavily specified code. Users should obtain benefits from writing richer specifications rather than be penalized.

[5] ESC/Java2, http://secure.ucd.ie/products/opensource/ESCJava2/

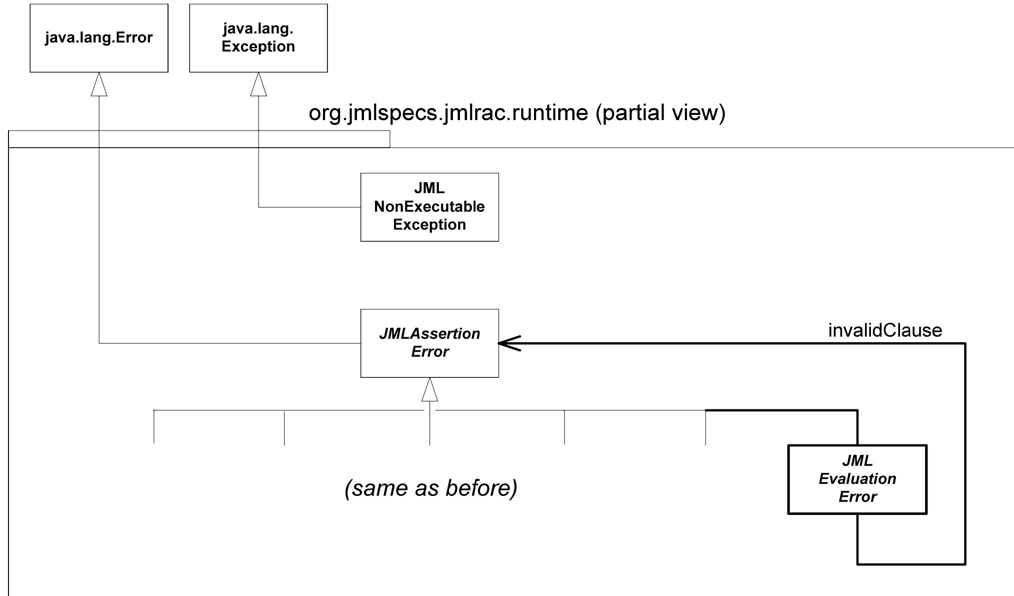

Figure 3: Modified error hierarchy of the JML RAC runtime package

4 JML RAC Redesign

4.1 Expression Evaluation

4.1.1 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. This overcomes the practical limitation presented earlier.

4.1.2 Original Expression Evaluation

Expression-evaluation code is generated by the RAC expression translator. The most general expression translator visits the AST nodes and generates code to evaluate the expression. Instances of expression translators are used by higher-level translators. The expressions to be evaluated by JML RAC-compiled code are part of statements. Each such statement has its own translator. Three higher-level translators use the basic expression translator. These three translators generate invariant assertion methods.

The higher-level translators use the expression translator in the following way: They instantiate the expression translator by providing it with, among others, the expression they want to be translated. They then use it to get a compound "node" that contains the code that evaluates the expression. Such a node can either be wrapped with a try-catch block or not. The high-level translator uses that node whenever it needs for that expression to be evaluated while generating the assertion methods.

The translation process is achieved by visiting every subexpression of the top-level expression and generating nodes to evaluate the subexpressions. As mentioned earlier, a new variable is usually defined to hold the value of a subexpression. Each of these nodes is stored in

a stack. When all of the expressions have been visited, the stack of nodes is used to generate an encompassing node that, if desired, can be wrapped in a try-catch block before being returned to the sender.

Quantified expressions are a special case. They require some static analysis in order to decide how, if possible, to evaluate them. The expression translator translates them using a helper. The approach to the evaluation of quantified expressions is discussed in Section 4.2.

4.1.3 New Approach to Expression Evaluation

The implementation of the new semantics requires alternate expression translators. For this reason, we have created a new general expression translator. This translator can be used wherever the old one was used in the past.

In the new approach to expression evaluation, there is no need to evaluate subexpressions separately through the use of newly declared variables. Precedence of operators is embedded in the AST, hence, a clever use of parentheses while visiting the tree allows this expression evaluation approach not to require any new variables. Since in the new semantics, a clause is either entirely executable or not, 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:

- non-executable exceptions discovered at runtime and
- all other exceptions.

Runtime non-executable exceptions cause the entire assertion to fold to a user-defined value (see Section 2.3.1). Any other exception or error thrown while evaluating an expression indicates that the expression being evaluated is, at least in part, invalid. Unlike the previous semantics, no truth value is assigned to the clause; instead, a new error is thrown: the `JMLEvaluationError`. The `JMLEvaluationError` solely means that the assertion is invalid. It also contains a reference to one of the other error classes in order to record information on the type of clause that failed, as illustrated in Figure 3.

4.2 Handling Quantified Expressions

Quantified expressions, unlike the 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 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 was used in the evaluation of the assertion in lieu of the quantified expression as described in [Rioux06].

Figure 4 illustrates how inner classes are used in code generation. It shows an excerpt of a class that features a method whose precondition is a clause containing a quantified statement (1). Below that, the code generated to evaluate the precondition is presented. The inner class's body and instantiation is highlighted in (2). Part (3) shows the evaluation string and (4) shows where the inner class being used to evaluate the quantified expression. The method of the inner class will only be called if the evaluation does not short-circuit.

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

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

(1)

```
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();
```

(2)

```
   rac$pre4 = (((x > y) || (x < 0)) || rac$v0Evaluator.eval());
```

(3) (4)

```
} catch (JMLNonExecutableException rac$v5$nonExec) {
   rac$pre4 = true;
} catch (Throwable rac$v6$cause) {
   JMLChecker.exit();
   throw new JMLRacExpressionEvaluationError("Invalid Expression in \
"MyClass.java\", line 36, character 10", rac$v6$cause);
}
```

Figure 4, Usage of inner classes for code generation of quantified expressions checking

5 Validation: Assessment and Statistics

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

```
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, character 20", rac$v1$cause);
8| }
```

Figure 5: Evaluation of precondition in the modified RAC

5.2 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 consist of more than 450 files and make extensive use of the features of JML.

Such a test suite helped us discover some flaws that occurred in rare circumstances. Most of them regarded situations where operator precedence is not preserved during the translation from JML to Java. The AST provided by the JML checker already has the precedence in it. The old JML RAC did not need to be aware of operator precedence as it is embedded in the AST. However, by combining many subexpression evaluations into one string, our new semantics had to generate parentheses, under some situations, in order to preserve JML operator precedence.

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 our code generation.

5.3 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 with the JML RAC demonstrated that it would fail to compile about a few source files. 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 did compile using both semantics, we gathered statistics to measure the overall reduction in code size under the new approach.

5.4 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. 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 code of Figure 2 (59 LOC) would be replaced by the one showed in Figure 5 (8 LOC).

The aim of this research was not to measure execution speed but rather to propose a new semantics. Nonetheless, 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 ESCTools, both the generated instrumented source code and bytecode showed a significant size reduction, as illustrated in Table 3. For instance, for

Table 3: ESCTools' escjava and javafe package statistics

	Original RAC Semantics	New RAC Semantics	Δ	New/Original Ratio
Escjava Instrumented source code size (MB)	33.6	26.5	7.1	≈ 78.9%
Escjava Instrumented bytecode size (MB)	12.2	9.8	2.4	≈ 80.3%
Javafe Instrumented source code size (MB)	35.5 *30.5*[*]	21.7 *21.6*[*]	13.8 *8.9*[*]	≈ 61.1% ≈ *70.8%*[*]
Javafe Instrumented bytecode size (MB)	10.7	8.0	2.7	≈ 74.8%

[*] denotes a corrected measurement

```
//@ public model boolean _isPrimitive; // cf. _isArray
//@         represents _isPrimitive = isPrimitive();

//@ ensures \result == _isPrimitive;
public /*@ pure @*/ native boolean isPrimitive();
```

Figure 6. Error in java/lang/Class.jml specification

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 the instrumented bytecode, the new/original semantics ratio was of 80.5%.

The Java front end of ESCTools (javafe package, 216 classes) displayed similar improvements in size. Table 3 presents the numbers. The practical limitations earlier described in this paper affected two classes in javafe. If we factor out those two abnormally large and non-compilable instrumented source files, the new/original size ratio goes from 61.1% to 70.8%. Such a modification is presented as a correction in the table.

6 Conclusion and Future Work

This work 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 [Chalin07b, §6.1]. We have as yet to uncover any new bugs using the RAC under the new semantics, but we have demonstrated that the RAC instrumentation code is much more compact and slightly more efficient. In fact, it is compact enough that the RAC can now be used to compile ESC/Java2 (this was previously impossible under the old assertion semantics).

We are currently considering extending the definedness checking of the RAC so that it will be able to detect invalid specifications like the one given in Figure 6. The problem is that a model field is defined in terms of the specification of the `isPrimitive()` method, which in turn is defined in terms of the original model field.

References

[Chalin05] P. Chalin. *Reassessing JML's Logical Foundation*. In Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP'05), Glasgow, Scotland, July, 2005.

[Chalin06] P. Chalin. *De-risking the Verifying Compiler Project: Recovering Soundness.* Technical Report ENCS-CSE-TR 2005-009, Dept. of Computer Science and Software Engineering, Concordia University, 2006.

[Chalin07a] P. Chalin. *Are the Logical Foundations of Verifying Compiler Prototypes Matching User Expectations?* Formal Aspects of Computing, 2007.

[Chalin07b] P. Chalin. *A Sound Assertion Semantics for the Dependable Systems Evolution Verifying Compiler*. In Proceedings of the International Conference on Software Engineering (ICSE), Minneapolis, MN, USA, 2007.

[Cheon03] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. Department of Computer Science, Iowa State University, TR #03-09, April 2003.

[Cheon+05] Y. Cheon, G. T. Leavens, *A contextual interpretation of undefinedness for runtime assertion checking*. In Proceedings of International Symposium on Automated Analysis-Driven Debugging, 2006.

[Dai05] K. Dai. *Enhancements to the JML Runtime Assertion Checker Compiler*. M.Comp.Sc. thesis, Concordia University, Montreal, Quebec, 2005.

[Gries+04] D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1994.

[Gries+05] D. Gries and F. B. Schneider. *Avoiding the undefined by underspecification*. In Jan van Leeuwen, editor, Computer Science Today: Recent Trends and Developments, number 1000 in Lecture Notes in Computer Science, pages 366-373. Springer-Verlag, New York, NY, 1995.

[Leavens+05] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. *How the design of JML accommodates both runtime assertion checking and formal verification*. Science of Computer Programming, Volume 55, pages 185-205, Elsevier, 2005.

[Leavens+06] G. T. Leavens, Y. Cheon. *Design by Contract with JML*, draft paper available from www.jmlspecs.org.

[Lindholm+99] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Prentice Hall, 1999.

[Rioux06] F. Rioux, *Effective and Efficient Design by Contract for Java*. M.Comp.Sc. thesis, Concordia University, Montreal, Quebec, 2006.