Reassessing JML's Logical Foundation

Patrice Chalin

Dept. of Computer Science and Software Engineering,
Dependable Software Research Group, Concordia University
www.cs.concordia.ca/~chalin

Abstract. Early in the design of the Java Modeling Language (JML) care was taken in the choice of its logical foundation to ensure that JML could accommodate run-time assertion checking, static analysis and formal verification. At the time, classical two-valued logic was adopted. Since then however, we note that the main JML tools have actually implemented differing semantics, by design. In this paper, we begin by reviewing the current logical semantics of JML and explore some of the ramifications of this choice. We then present the results of a survey of programmers from industry, i.e. JML's targeted end users. We asked them how they want assertions to be interpreted during run-time checking and static verification. Survey results indicate that developers are in favor of a semantics for assertions that is compatible with their current use in run-time checking, and hence consistent with a three-valued logic in which partial functions are modeled explicitly.

Keywords: assertions, run-time checking, verification, logical foundations, industry survey, Java Modeling Language, three-valued logic.

1 Introduction

The Java Modeling Language (JML) is a behavioral interface specification language for Java that can be used by developers to accurately document detail design decisions directly in their code [LBR99, Leavens+05]. As an added benefit to annotating their code in this way, developers can make use of several tools that process JML annotations. The tools offer a range of functionality: e.g. the creation of Javadoc-like documentation that includes the relevant JML annotations, instrumentation of code with run-time checking of assertions, automatic static checking of code against its specification, and even full formal verification. The last three of these capabilities are supported by the JML run-time assertion checker compiler [Cheon03], ESC/Java2 [ESCJ2] and the LOOP tool [JP03], respectively.

Early in the design of JML care was taken in the choice of its logical foundation to ensure that JML could accommodate the contending needs introduced by run-time assertion checking (RAC), static analysis and formal verification. The main challenge was—and still is—dealing with undefined expressions in assertions as introduced by partial functions, e.g. 4/0 or `a[0]` when `a` is null. The goal was to opt for a logic that would be suitable to "programmers and mathematicians" alike [Leavens+05, §1.3.2], and the consensus that was arrived at was a two-valued

logic in which partial functions are modeled by underspecified total functions [GS95].

Since then however, we note that the main JML tools have actually implemented differing semantics, and this by design. In this paper we review the current logical semantics of JML and explore some of the ramifications of this choice. We examine why one of the tools, the run-time assertion checker compiler (RACC), can at best approximate the semantics.

In light of this situation we believe it is time to reassess JML's logical foundation. To set the stage for the reassessment, we present the results of a survey [Chalin05] whose participants were mainly programmers from industry—i.e. JML's targeted end users. In this survey, we asked developers whether they used assertions and what they used them for. We also presented them with questions whose answers can help guide us in choosing an appropriate logical foundation for JML.

1.1 Motivation

Writing trustworthy software is a challenge. Several research (and commercial) initiatives are underway whose common goal is to develop tools and techniques that will enable software engineers to write more dependable applications. One of these initiatives is the Dependable Systems Evolution Grand Challenge (DSE GC). An important component of the DSE GC is the verifying compiler (VC) project [DSEGC04]. Such a compiler is meant to be used to automatically prove that a program or program component is correct. Correctness is defined by *program assertions* (and other redundant annotations) that are judiciously placed in the application code [DSEGC04]. When unable to prove correctness, a VC may embed assertions in object code for the purpose of run-time checking, similar to what is done by today's compilers with respect to type checking.

DSE GC proposals stress that the tools and techniques that are to be developed should be targeted for use by *real programmers* writing code for *normal* commercial, industrial or open source software using *mainstream* languages [Woodcock03]. We believe that JML and its associated tools can be seen as early technological prototypes for a VC.

1.2 Importance

Like any other software engineering effort, the success of the DSE GC (and of JML) depends on the appropriate involvement of stakeholders, particularly end-users [LW03]. When the end-user base is large, it is particularly important to consult with a representative set of the population [Pressman01]. Failure to do so can significantly decrease the likelihood of user adoption.

Who are the targeted end users of this technology? Some JML references name mathematicians and programmers [Leavens+05]. DSE GC proposals and most JML literature clearly indicate that the main end users are general practitioners writing software for varied application domains—e.g. "the aim of JML is to provide a specification language that is easy to use for Java *programmers*"

[Burdy+05]. Hence, the importance of gathering the opinions of our main targeted end user group, and then using this to drive the reassessment.

1.3 Outline

In Section 2 we provide an introduction to the programmer survey following by a brief review of JML's current logical foundation (Section 3). Section 2 presents only enough of the survey results to motivate the discussion on the logical foundation that is given in Section 4. In Section 5 we share a summary of "what practitioners want" as a logical foundation for program assertions. We discuss these results (Section 6) and then conclude (Section 7).

2 Programmer survey, an introduction

We recently conducted an end-user survey of programmers, mostly from industry. The survey was open to all programmers, not only those developing Java applications though 63% of respondents reported programming in Java. The main purpose of the survey was to uncover the preferences of programmers with respect to the logical semantics of program assertions in the context of run-time checking and extended static checking (a form of static analysis).

Over two hundred developers participated in the survey, 77% worked in industry and 26% worked in academia. Most respondents were from the United States (46%), Europe (23%) and Canada (23%). Respondents worked in a variety of application domains such as: business/finance, entertainment, medical, military, security, software tools, and systems software. All but one respondent claimed that assertions were used at their institutions[1]. Respondents were asked to choose a representative product developed or maintained at their institution and to estimate the proportion of lines of code (LOC) that were assertions; on average between 1.4% and 5% was reported. Assertions are primarily used (97%) in run-time assertion checking (RAC). About 20% reported use of extended static checking (ESC) or static analysis (SA) tools.

3 JML's current logical foundations

JML is based on classical two-valued logic in which partial functions are modeled by underspecified total functions [GS95] as we explain next. Let *f* be a function in such a logic then *f* will be defined for all values of its domain and *f*(*v*) will always have a value in the range of *f*. If *f* is usually undefined at *v* then *f*(*v*) will have some unspecified value in the range of *f*. As an example, consider the array access expression `a[0]` in the context of the following declaration:

```
int a[] = null;
```

In this case, `a[0]` is undefined because `a` is null. In this logic, `a[0]` will have some integer value, although we do not know which value it is. Note that `a[0]`

[1] Note that the purpose of the survey was not to perform a random sampling of developers so as to determine the proportion that actually made use of assertions. We are simply noting here that of the responses received, only one claimed not to have used assertions.

and `a[1]` are not necessarily equal since the array access operator is being applied to different arguments, (null,0) and (null,1) respectively, and hence these might be mapped to different values.

One of the advantages of modeling partial functions in this way is that the rules of classical logic with equality can be preserved. Thus, in particular

- equality remains reflexive so that `a[i] == a[i]` regardless of the values of `a` and `i`, and
- the law of excluded middle holds, e.g. `a[i] == 1` or `a[i] != 1` for any value of `a` and `i`.

In the remainder of this article, we will use the term "classical two-valued logic" (or simply classical logic) to mean "two-valued logic with partial functions modeled as underspecified total functions".

As a consequence of having adopted a two-valued logic, Java's conditional Boolean operators naturally become equivalent to their non-conditional counterparts; i.e. conditional conjunction (`&&`) and conditional disjunction (`||`) are interpreted in the same way as Java's Boolean non-conditional[2] conjunction (`&`) and disjunction (`|`), respectively. Thus,

```
a != null && a[0] > 1
```

is taken to be logically equivalent to

```
a != null & a[0] > 1
```

and

```
a[0] > 1 & a != null
```

(which may come as a surprise to practitioners).

4 Classical logic and run-time assertion checking

4.1 Approximating classical logic, at best

Classical two-valued logic cannot be practically implemented in run-time assertion checking code. The main challenge is that for any given function *f* and arguments *v* that are outside the domain of *f*, the system must choose an arbitrary value for *f*(*v*) and it must record this value so that it can be returned in all subsequent cases where *f* is applied to *v*. Note that by a function *f* we mean any JML operator or (functional) pure method. Pure methods are the only methods that can be used in assertion expressions.

The JML run-time assertion checker compiler (RACC) does its best to *approximate* classical logic. It does so by interpreting an atomic undefined proposition as either true or false depending on the context [Cheon03, §3.6]. Assertion evaluation is treated as a game where the system tries to maximize its chances at winning (i.e. making an undefined assertion false at the top-level)

[2] Non-conditional operators are called "logical operators" in Java [JLS2, §15.22].

while keeping false positives to a minimum [CL05][3]. Hence the RACC would choose to make `a[0] > 0` false in

```
a[0] > 0 | true
```

but it would make it true in

```
!(a[0] > 0) | true
```

Note that in both cases the overall expressions will be true because the second disjunct is true.

The contextual approach does have some "anomalies" (as they are called by the RACC author). For example, let `a` and `b` be array references that are null, then `a[0] == b[0]` is interpreted as false but then again so is `a[0] == b[1]`. In classical logic the former would be true and the latter undetermined. To deal with this situation the RACC author has chosen to depart from the JML semantics and to report these kinds of assertions as exceptional assertion failures. The justification for this choice was for the run-time checking code to be of maximum benefit to programmers by catching potential errors due to undefinedness [Cheon03, §1.3.2]. Hence, in conclusion, the RACC does not implement a semantics of assertions based on classical logic.

4.2 Loss of referential transparency

It should be noted that the contextual approach leads to the loss of referential transparency. For example, let `f(int v)` be a `boolean` method with body

```
return v > 0 || true
```

then the following two assertions, though logically equivalent, would yield true and false (failure), respectively, in the RACC: `a[0] > 0 || true` and `f(a[0])`. Loss of referential transparency has the RACC depart even further from classical logic.

4.3 Industry

As was stated in Section 2, current industrial use of assertions is mainly for run-time assertion checking, and this is likely to remain the case well into the next decade. Thus, JML RAC support must adequately meet the needs of industry. It is unlikely that developers will accept an implementation of RAC that only approximates the intended assertion semantics. More importantly, the majority of developers have stated that they want RAC and static analysis tools to agree on their interpretation of assertions; details are given in Section 5.3. Hence JML's logical foundation needs to be reassessed if we want to meet the needs of our end users. What better way to ensure that we can meet those needs than by asking them?

[3] Another aspect of the "game" has to do with attempting to make top-level assertions true if they contain informal assertions. Since this feature is not relevant to our presentation, we do not discuss it here.

Table 1, Responses to Questions C.2, D.1, D.4

Question	true	false	error/ exception	other
C.2	0%	9%	**81%**	9%
D.1 <EXPR>				
t || (nullRef[0] > 0)	**73%**	1%	18%	7%
(nullRef[0] > 0) || t	6%	5%	**84%**	4%
nullRef[0] > 0	1%	5%	**91%**	3%
t | (nullRef[0] > 0)	4%	5%	**87%**	3%
D.4 <EXPR>				
a[0] == a[0]	16%	7%	**75%**	3%
a[0] == b[0]	7%	8%	**80%**	4%
a[0] == b[1]	3%	8%	**84%**	5%
g(a[0]) == a[0]/a[0]	6%	9%	**82%**	3%
a[0] == 0 || a[0] != 0	8%	10%	**74%**	7%

5 Logical foundations: what do practitioners want?

A summary of the survey results is given in this section. Further details can be found in [Chalin05].

5.1 Exceptions during RAC

We asked:

> **C.2**. What should be done during **run-time assertion checking** if an error/exception is reported during the evaluation of an **assertion expression** (such as a[0] > 0 when a is null)?

The choice of answers was:

- Interpret the expression as **true**.
- Interpret the expression as **false**.
- Report an **error/exception**.
- Other (provide details).

5.2 Exceptions and ESC

We presented various expressions and asked respondents how they should be interpreted in the context of static analysis (SA).

> **D.1**. During **static analysis** how should each of the following assertion expressions, <EXPR>, be interpreted when **nullRef** is a **null** array reference and, **t** is **true**?

The <EXPR>s along with the profile of responses are given in Table 1. The first expression involves a conditional-or operator with its first argument being true. In such a case the value of the second argument is not interpreted and the overall

expression evaluates to true. The majority of respondents chose "true". The next most popular answer was "error/exception" which appears to have been chosen, in some cases, because the reader was unaware that "||" is a conditional-or in C-based languages[4]. Hence we see that developers would prefer that JML preserve the semantics of conditional operators.

All other expressions in **D.1** would result in a null pointer exception if evaluated during run-time checking. Most respondents were in favor of the same interpretation in the case of static analysis.

The last expression of **D.1** uses Java's non-conditional or; the first term is true and the second undefined. We see that most developers prefer that the interpretation of non-conditional operators be *strict* (also called Weak Kleene).

5.3 Consistency in the interpretation of assertions

Respondents were also asked whether RAC and SA tools should be consistent in their interpretation of any given assertion (and for a given specific program state).

> **D.2**. For any given assertion expression *E* and program state *S*, should **run-time assertion checking (RAC)** and **static analysis (SA)** always agree on the same interpretation of *E*? That is, should
>
> - they *both* interpret the assertion as **true**, or
> - they *both* interpret it as **false**, or
> - RAC will report an **error/exception** and SA will interpret the assertion as being in error.

73% of respondents answered "yes" citing:

- "[consistency avoids] special cases; helps me remember how things work."
- "Anything other than complete agreement will inevitably lead to confusion",
- and a few commented that to do otherwise would violate the principle of least surprise.

Several who answered "no" did so because they remarked that SA might not be able to determine the value of arbitrary assertions. This is true in general, hence we were careful to phrase the question in terms of a given program state *S*. Some respondents appear of have missed this point.

5.4 Questions specific to classical two-valued logic

We anticipated that the respondents who answered no to **D.2**, might have in mind the modeling of partial functions by underspecified total functions. To test this hypothesis we asked such respondents to complete the following question:

> **D.4**. During **static analysis** how should each of the following assertion expressions, <EXPR>, be interpreted when **a and b are null**?

The <EXPR> of **D.4** were chosen so as to highlight issues that might arise under the interpretation of partial functions.

[4] A small proportion of developers did not program in C-based languages.

As indicated in Table 1, 16% of respondents believed that `a[0] == a[0]` should be true when `a` is null—consistent with an interpretation in two-valued logic. On the other hand `a[0] == b[0]` would also be true under a two-valued logic when `a` and `b` are null because the expression simplifies to `null[0] == null[0]` which is true; yet only 7% of respondents recognized this. As was explained in Section 3, the third expression would have an undetermined value in a two-valued logic because the arguments to the array access operator are different. The next expression involves a function `g` declared as follows:

```
int g(int m) {
  return m/m;
}
```

It is unclear how the expression `g(a[0]) == a[0]/a[0]` should be interpreted in classical logic. An issue of concern is the following: since `a[0]`—the argument to `g`—is undetermined, should we still attempt to interpret `g` at `a[0]`? The final expression is an instance of the law of excluded middle—a central axiom of two-valued logic[5]. The majority of respondents chose "error/exception" for all expressions.

6 Discussion

6.1 What practitioners want

In the main paper discussing the choice of logical semantics for JML it is written "we are willing to accept a slightly different semantics [from Java's semantics] for assertion evaluation as long as programmers are not too surprised by it" [Leavens+05, §1.3.2]. It would appear that developers would be surprised by the current JML semantics.

The survey results indicate that practitioners are in favor of an interpretation of assertions, be it for run-time or extended static checking, in which an error/exception is reported when a partial function is applied to arguments outside its domain. In the next sections, we explore some of the top factors that may have influenced this choice.

6.2 Why they want it

- Two-valued logic is misaligned with programming practice.

  The laws of classical logic do not hold in the context of most programming languages. For example, `1/x == 1/x` will not be interpreted as true if `x` is 0, instead a "division by 0" exception will be raised. Exceptions raised under such circumstances signal the presence of an error in the program; e.g. the programmer must have believed that `x` could not be 0 if `1/x` was to be evaluated (of course the belief could have been wrong; in either case a bug has been exposed). Exceptions have been recognized as a useful tool in helping to detect such programming errors as close to their point of occurrence as possible.

[5] It is amusing to note how the second most popular answer was false rather than true. (Of course the differences between the number of responses for true and false is not statistically significant.)

It is obvious that the evaluation of program expressions can result in side-effects, thus easily contravening the laws of logic. It is well understood by programmers that assertions, as well as any debugging code, must be free of side-effects. Of course, any assistance by languages and tools in preventing and/or detecting potential side-effects would be welcome.

- Ignoring errors/exceptions is *bad* programming practice.

  Rephrased in programming terms, modeling a partial function as an underspecified total function, essentially amounts to catching exceptions raised by the partial function and ignoring or masking them by returning an arbitrary legal value. This makes it much more difficult to locate the origin of an error. It is also known to be bad programming practice, e.g. Item 47 in [Bloch01], "*Don't ignore exceptions*".

- Consistency in the interpretation of assertions across tools

  As was reported in Section 5.3, the top reason for having run-time checking and static analysis agree on the interpretation of assertions is consistency. Lack of consistency would require that practitioners be versed in *two* logical systems. Managing one logical system is already a challenge for the majority of practitioners and students—as is exemplified by ongoing debates on the role of mathematics in computer science and software engineering education—e.g. [Devlin03]. The need to learn and use two logical systems will have an impact on costs, e.g. due to training, and productivity. The impact on productivity should be apparent to anyone who has developed software in two *mostly similar* but *subtly different* languages—e.g. C++, Java or C#. The minor differences in language semantics often gives rise to subtle bugs.

  Of course, consistency could be achieved by adapting the semantics of run-time checkers to conform to a two-valued logic of static checkers but such an adaptation can only be approximated, for all practical purposes (see Section 4.1). It seems more sensible, from a practitioners point of view, for the semantics of static checkers to be adapted to conform to the semantics of run-time checkers.

6.3 Logics

The survey results indicate that the majority of practitioners would prefer a logic in which partial functions and undefinedness were modeled directly in a manner that is consistent with the operational semantics of programming languages. Three-valued logics appear more suitable in this case.

Barringer, Cheng and Jones have explored various formulations of three-valued logic [BCJ84], finally settling upon what has become known as the Logic of Partial Functions (LPF), the logic underlying the Vienna Development Method (VDM) [Jones90]. LPF adopts a choice of logical operators (called Strong Kleene) that are non-strict and monotonic [CJ91]. While such a choice of operators may be suitable for a foundation of LPF, strict and conditional operators will also be required if we are to accurately reason about the logical connectives of Java (cf. Section 5.2).

As pointed out by Cheng and Jones, conditional operators enjoy fewer properties (such as commutativity and distributivity), but this is a cost that end users may need to experience first-hand: e.g. if the use of non-conditional operators (such as Java's "|") allows ESC tools to prove more properties automatically, then practitioners may be more inclined to use them. Habits will not be changed unless there is sufficient motive to do so. In a separate survey, we have noticed that Eiffel programmers, for example, make use of non-conditional operators more frequently than their conditional counter parts [Chalin05b]. This may stem from the fact that Eiffel conditional operators—whose syntax is borrowed from Ada—are longer to write: e.g. "or else" vs. simply "or". No matter what the reason, the end result may well be that it will be easier to (automatically or manually) verify the validity of Eiffel contracts than, say, JML contracts. In the end, the use of conditional operators in programs will not disappear. Hence we will need a logic suitable for reasoning about them.

The survey results support an interpretation of assertions that is consistent with a three-valued logic. Of course, no statement is being made concerning the necessity of using a three-valued logic in the provers underlying verification tools. It is well known that two-valued logics are sufficient to model three valued logics—e.g. [Konikowska93, JM94].

7 Conclusion

Every experienced programmer knows that exceptions report *exceptional* situations and that it is bad programming practice to ignore them [Bloch01]. This holds true for program assertions as well (i.e. exceptions raised by partial functions should not be ignored during assertion interpretation) yet, this is what the current JML semantics propose. We have shown how, for all practical purposes, JML's logical semantics cannot be consistently implemented by both RAC and ESC tools. Our programmer survey confirms that developers want tools to adopt a common semantics. Furthermore, they are in favor of a semantics that is consistent with their current experiences with run-time assertion checking. We note that the semantics of JML assertions could naturally be based on a three-valued logic—e.g. a suitably extended LPF.

References

[BCJ84] H. Barringer, J.H. Cheng, and C.B. Jones, "A Logic Covering Undefinedness in Program Proofs," *Acta Informatica*, vol. 21, no. 3, pp. 251-269, 1984.

[Bloch01] Joshua Bloch. *Effective Java Programming Language Guide*. Addison-Wesley. The Java Series. 2001.

[Burdy+05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino and Erik Poll. An overview of JML tools and applications. In *International Journal on Software Tools for Technology Transfer* (STTT), vol. 7, no. 3, June 2005.

[Chalin05] P. Chalin. "Logical Foundations of Program Assertions: What do Practitioners Want?" In *Proceedings of the 3rd International Conference on Software Engineering and Formal Methods*, Koblenz, Germany, September 5-9, 2005 (to appear).

[Chalin05b] P. Chalin. "Are Practitioners Writing Contracts?" In *Proceedings of the Workshop on Rigorous Engineering of Fault Tolerant Systems* (REFTS'05), July 2005 (to appear).

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

[CJ91] J.H. Cheng and C.B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J.C.P. Woodcock (eds), *3rd Refinement Workshop*, pp. 51-69. Springer Workshops in Computing Series, 1991.

[CL05] Yoonsik Cheon and Gary T. Leavens. *A Contextual Interpretation of Undefinedness for Runtime Assertion Checking*. Department of Computer Science, University of Texas at El Paso, TR #05-10, March 2005.

[Devlin03] Keith Devlin. Why Universities Require Computer Science Students To Take Math. *CACM* 46(9):36-39, September 2003.

[DSEGC04] GC6 Steering Committee. *Verified Software Repository: a step towards the Verifying Compiler*. Draft, 2004-12-20.

[ESCJ2] ESC/Java2 site at secure.ucd.ie/products/opensource/ ESCJava2.

[GS95] D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, LNCS 1000, pp. 366–373. Springer-Verlag, 1995.

[JLS2] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 2nd ed. Addison-Wesley Professional, 2000.

[JM94] C.B. Jones and C.A. Middelburg, "A Typed Logic of Partial Functions Reconstructed Classically," *Acta Informatica*, vol. 31, no. 5, pp. 399-430, 1994.

[Jones90] C.B. Jones. *Systematic Software Development using VDM*. Computer Science Series. PHI, 2nd ed., 1990.

[JP03] B. Jacobs and E. Poll, "Java Program Verification at Nijmegen: Developments and Perspective." In *Proceedings of the International Symposium on Software Security - Theories and Systems (ISSS 2003)*, LNCS 3233, pp. 134-153, 2003.

[Konikowska93] B. Konikowska, "Two Over Three: A Two-Valued Logic for Software Specification and Validation Over a Three-Valued Predicate Calculus," *Journal of Applied Non-Classical Logics*, vol. 3, pp. 39-71, 1993.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. "JML: A notation for detailed design." In H. Kilov and B. Rumpe, editors, *Behavioral Specifications of Business and Systems*, pp. 175-188. Kluwer, 1999.

[Leavens+05] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. "How the design of JML accommodates both runtime assertion checking and formal verification." *Science of Computer Programming*, vol. 55, pages 185-205, 2005.

[LW03] Dean Leffingwell and Don Widrig. *Managing Software Requirements: A Use Case Approach*. Second edition. Addison-Wesley, Object Technology Series, 2003.

[Pressman01] Roger S. Pressman *Software Engineering: A Practitioner's Approach*, 5th edition. McGraw Hill. 2001.

[Woodcock03] J.C.P. Woodcock. *Dependable Systems Evolution: A Grand Challenge for Computer Science*. 2003-05-26.