Formal Semantics for the Java Modeling Language

Daniel Bruns
Karlsruhe Institute of Technology
bruns@kit.edu

A common critique of formal methods in software development practise is, that they are not readily understandable and thus not widely used (see for instance [Nam97]). The Java Modeling Language (JML) was created in an attempt to bridge that gap. By building upon the syntax of Java it is meant to be easily accessible to the common user – who might *not* be skilled in formal modeling. Due to this advantage, JML has quickly become one of most popular specification languages to use with both static and runtime analysis of programs. JML specifications are written in a Java-like expression language as comments straight into source files. It provides both in-code assertions as well as method contracts and class invariants which are indicated by appropriate preceding keywords. Fig. 1 shows a method specification. However, the official reference [LPC$^+$09] mostly lacks a clear definition of semantics. In turn, several tools implementing JML syntax use their own interpretations. Past approaches to formal semantics have rather been documentations of those tools than providing a unified reference.

```
/*@ requires x >= 0;
 *@ ensures \result == (\product int z;
 *@                        0 < z & z <= x; z); @*/
public int factorial (int x) {
  if (x <= 0) return 1;
  else return x * factorial(x-1);
}
```

Figure 1: Method specification with pre- and post-condition for a factorial calculation. Note that it is only partial; the result of negative inputs is not specified. Special expressions for the method result and a product over integers are used.

The present work makes use of purely mathematical notations, such as predicate logic with sets and functions, to describe a simple machine model which provides the semantical entities for a rigorous evaluation of JML artifacts. Predominantly, the semantics given throughout this work are based on the informal descriptions found in the JML Reference Manual. As many features are defined in terms of Java, we relied on the Java language specification [GJSB05] where needed – without formalizing Java itself. As a result, most structures which are commonly used in the specification of sequential Java are covered by this formalization. As this work had revealed several conceptual flaws or inconsistencies within the language design, we have put further efforts in pushing a sound official documentation.

Foundations First of all, we define a formalization of the virtual machine on which a Java program runs. A (closed) program defines a *universe* $\mathcal{U}$ of primitive values and typed semantical objects. We define a *system state* $s \in \mathcal{S}$ as a triple (h, σ, χ) where (i) $h : \mathcal{U} \times \mathcal{I} \to \mathcal{U}$ is a function representing the *heap* by mapping pairs of objects and identifiers to elements of the universe, (ii) $\sigma : \mathcal{I} \to \mathcal{U}$ is a function representing the *stack* by evaluating local variables and (iii) χ is a sequence of named methods and respective receiver objects representing a *call stack*. An executable fragment of a program (say, a method body) invoked in a pre-state s_0 yields a (possibly infinite) sequence R (the *run*) of states which are passed through in execution, a termination witness Ω (an instance of `Throwable` where `null` denotes a successful execution) and a return value ρ.

Expressions The syntax of JML expressions is essentially the same as in Java, but enriched with quantification and special constructs, such as `\result` to access the return value of a method invocation, or `\old` to refer to pre-state values of expressions (in a method's post-state). We use an evaluation function $val_{s_0}^{s_1} : Expr \to \mathcal{U}$ which depends on two system states – a pre-state s_0 and a post-state s_1. Formulae in this sense are just `boolean` typed expressions. Several features (adapted from Java) however, make JML expressions harder to handle than classical first order logic: (i) Expressions may have side-effects under certain circumstances, e.g. `new Object() == o` is a legal assertion. This means for expressions not only yielding a value but also a transition from one state to another. We describe this using a function $\omega : Expr \to \mathcal{S}^{\mathcal{S}}$. As a result, for *any* expression the order of sub-expressions does matter. (ii) The exceptional behavior of Java is imitated, which can be interpreted as a covert third truth value. E.g. `0/0 == 0/0` is not semantically valid, even though it appears to be a logical tautology. In order for a boolean expression be considered *valid*, it has not only to yield the value $true$ but also be well-defined. This is evaluated by the function $wd_{s_0}^{s_1} : Expr \to \{true, false\}$. (iii) To the above two items *short-circuit* evaluation applies as in Java. As an example, a boolean disjunction is evaluated in the following way:

$$\begin{array}{lllll} val_{s_0}^{s_1}(A \parallel B) = true & \text{iff} & val_{s_0}^{s_1}(A) = true & \text{or} & val_{\omega(A)(s_0)}^{\omega(A)(s_1)}(B) = true \\ wd_{s_0}^{s_1}(A \parallel B) = true & \text{iff} & wd_{s_0}^{s_1}(A) = true & \text{and} & \\ & & val_{s_0}^{s_1}(A) = true & \text{or} & wd_{\omega(A)(s_0)}^{\omega(A)(s_1)}(B) = true \end{array}$$

As a result, `true` $\parallel 0/0 == 0/0$ is a valid proposition, while its commutation is not. While this distinction is perfectly reasonable from a programmer's view, additional effort has to be taken to incorporate this into classical logic calculi. Fortunately, the definition val is independent of wd. This offers the possibility to encode wd as a formula of classical logic and to view it as an additional proof obligation while leaving the definition of truth untouched.

Specifications JML extends the design-by-contract paradigm [Mey92] by introducing method contract facilities which allow not only to specify pre- (`requires` α) and post-conditions (`ensures` β) for successful executions, but also post-conditions for exceptional termination cases (`signals` (e) γ) as well as necessary pre-conditions for nontermination (`diverges` δ). Assuming the pre-condition to hold[1] in state s_0 (i.e. $val^{s_0}_{?}(\alpha) = true$), there are different conditions to hold depending on the mode of termination (with the above definition of R, Ω and ρ):

Termination mode	Necessary condition
None ($\vert R\vert = \infty$)	$val^{s_0}_{?}(\delta) = true$
Exceptional ($R = \langle s_0, \ldots, s_k\rangle, \Omega \neq \texttt{null}$)	$val^{s_k \oplus \{e \mapsto \Omega\}}_{s_0}(\gamma) = true$
Normal ($R = \langle s_0, \ldots, s_k\rangle, \Omega = \texttt{null}$)	$val^{s_k \oplus \{\backslash\texttt{result} \mapsto \rho\}}_{s_0}(\beta) = true$

Invariants are evaluated in a similar way, but their semantics are based on the *visible state* paradigm [Poe97]. This essentially means that objects have to respect invariants as long as they are not the receiver of a currently active method invocation. This is a stronger requirement than the common *observable state* semantics, which means to assert the invariant just in a method's post-state. That may lead to invariant violations upon re-entrant method calls. It turns out that the property of a state being visible is not deducible from itself, but from the call stacks of the run as a whole. The characterization in this thesis has shown several inconsistencies in the very design of visible states, such as virtually any constructor would be unable to even establish an invariant.

Model fields JML also features the notion of *model fields* [CLSE05] which are not part of the program but added in specification for means of abstraction. Yet there has been no common understanding in what they should represent. One viewpoint is that model fields abstract from the current machine state. Another one is that they may refer to objects which are not referenced by the program itself, thus defining a definitional extension to the logic. In the former case, their use would be very similar to that of logical axioms, while in the latter one has to provide extended evaluation facilities (e.g. define a second heap which holds values to the model object's fields). In any way, specification features must not interfere with the actual program. As this is still a hot topic, significantly more work is needed to reach a reference definition.

Tool Support The KeY tool [BHS07] for formal verification of Java Card programs provides a JML interface (among others). JML specifications are translated into formulae of dynamic logic [Eng05]. As KeY implements only a subset for which sound formal semantics exist, its interpretation of the language comes very close to the anticipated (informal) meaning. As the need for treatment of a substantial part of Java (including generics, concurrency, etc.) rises, there is yet still a demand for more sound JML semantics.

[1] For simplicity, we only show val while wd has to hold for the same parameters. ? does denote an unspecified state here.

References

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[Bru09] Daniel Bruns. Formal Semantics for the Java Modeling Language. Diploma thesis, Universität Karlsruhe, June 2009.

[CLSE05] Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model Variables: Cleanly Supporting Abstraction in Design By Contract. *Software — Practice & Experience*, 35(6):583–599, May 2005.

[Eng05] Christian Engel. A Translation from JML to JavaDL. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, February 2005.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, 2005.

[LPC$^+$09] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. JML Reference Manual. Draft. Revision 1.235, Available from `http://www.jmlspecs.org/`, September 2009.

[Mey92] Bertrand Meyer. Applying “Design by Contract”. *IEEE Computer*, 25(10):40–51, October 1992.

[Nam97] Baudouin Le Charlier Namur. Specifications are Necessarily Informal or: The Ultimate Myths of Formal Methods. Technical Report BU-CEIS-9703, Bilkent University, Faculty of Engineering, 1997.

[Poe97] Arndt Poetzsch-Heffter. *Specification and verification of object-oriented programs*. Habilitationsschrift, Technische Universität München, January 1997.