How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification

Gary T. Leavens\(^1\), Yoonsik Cheon\(^1\), Curtis Clifton\(^1\), Clyde Ruby\(^1\), and David R. Cok\(^2\)

\(^1\) Department of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, Iowa 50011-1041 USA
\{leavens,cheon,cclifton,ruby\}@cs.iastate.edu
phone: +1 515 294 1580, fax: +1 515 294 1580

\(^2\) Eastman Kodak Company
Research & Development Laboratories
1700 Dewey Avenue, Building 65, Rochester, New York 14650-1816 USA
david.cok@kodak.com
phone: +1 585 588 3107, fax: +1 585 588 3269

Abstract. Specifications that are used in detailed design and in the documentation of existing code are primarily written and read by programmers. However, most formal specification languages either make heavy use of symbolic mathematical operators, which discourages use by programmers, or limit assertions to expressions of the underlying programming language, which makes it difficult to write complete specifications. Moreover, using assertions that are expressions in the underlying programming language can cause problems both in runtime assertion checking and in formal verification, because such expressions can potentially contain side effects. The Java Modeling Language, JML, avoids these problems. It uses a side-effect free subset of Java’s expressions to which are added a few mathematical operators (such as the quantifiers \(\forall\) and \(\exists\)). JML also hides mathematical abstractions, such as sets and sequences, within a library of Java classes. The goal is to allow JML to serve as a common notation for both formal verification and runtime assertion checking; this gives users the benefit of several tools without the cost of changing notations.

1 Introduction

The Java Modeling Language, JML \([55, 54]\), is the result of a cooperative, international effort aimed at providing a common notation and semantics for the specification of Java code at the detailed-design level \([58]\). JML is being designed cooperatively so that many different tools can use a common notation for Hoare-style behavioral interface specifications. In this paper we explain the features of JML’s design that make its assertions easily understandable by programmers and suitable for both runtime assertion checking and formal verification.
1.1 Background

By a Hoare-style specification we mean one that uses pre- and postconditions to specify the behavior of methods [31][33][44]. A behavioral interface specification language (BISL) is a specification language that specifies both the syntactic interface of a module and its behavior [31][48][52][81]. JML, the interface specification languages in the Larch family [31][48][52][81] and RESOLVE/C++ [22][73] are BISLs. Most design by contract languages and tools, such as Eiffel [70][71] and APP [77], are also BISLs, because they place specifications inside programming language code. By contrast, neither Z [80][79][87] nor VDM [6][27][74][43] is a BISL; they have no way to specify interface details for a particular programming language. OCL [82][83] is a BISL for the UML, but the UML itself is language-independent; this poses problems for a Java programmer, because the UML does not have standard notations for all details of Java method signatures. For example, the UML’s syntax for specifying the signatures of operations has no standard notation for declaring that a Java method is strictfp or for declaring the exceptions that a method may throw [7, pp. 128-129] [49, p. 516]. Also the OCL has no standard constraints that correspond to JML’s exceptional postconditions. Because BISLs like JML specify both interface and behavior, they are good at specifying detailed designs that include such Java details. This makes JML well suited to the task of documenting reusable components, libraries, and frameworks written in Java.

1.2 Tool Support

Because BISLs are easily integrated with code, they lend themselves to tool support for activities related to detailed design, coding, testing, and maintenance. An important goal of JML is to enable a wide spectrum of such tools. Besides tools that enforce JML’s semantics (e.g., type checking), the most important JML tools help with the following tasks.

Runtime checking and testing. The Iowa State group provides (from www.jmlspecs.org):

- the jmlc runtime assertion checking compiler [13], which generates class files from JML-annotated Java sources, and
- the jmlunit tool [14], which uses the runtime assertion checker to generate test oracle code for JUnit tests.

Documentation. David Cok provides the jmldoc tool, also available through www.jmlspecs.org, which generates HTML documentation similar to that

---

1 Larman notes that the UML has some nonstandard ways to specify the exceptions that a method may throw, by either using Java’s own syntax directly or by using a “property string”.

2 Besides this runtime assertion checking work at Iowa State, which relies on adding instrumentation to compiled code, Steven Edwards’s group at Virginia Tech is working on a wrapper-class based approach to runtime assertion checking that will allow instrumentation of programs for which source code is not available.