Lessons from the JML Project

Gary T. Leavens and Curtis Clifton

TR #05-12a
April 2005, Revised July 2005

Keywords: verifying compiler, specification, verification, formal methods, formal interface specification, extensible compilers, integrated development environments, JML language, Java language, Hoare

2000 CR Categories: D.2.1 [*Software Engineering*] Requirements/ Specifications — languages, tools, JML; D.2.2 [*Software Engineering*] Design Tools and Techniques — computer-aided software engineering (CASE); D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, formal methods, programming by contract, reliability, tools, validation, JML; D.2.5 [*Software Engineering*] Testing and Debugging — Debugging aids, design, testing tools, theory; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques.

Submitted for publication

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Lessons from the JML Project

Gary T. Leavens[1] and Curtis Clifton[1]**

Department of Computer Science, Iowa State University,
226 Atanasoff Hall, Ames, IA 50011 USA

Abstract. To have impact, a grand challenge should provide a way for diverse research to be integrated in a synergistic fashion. Synergy in the JML project comes from a shared specification language, and thus holds several lessons for the verifying compiler grand challenge. An important lesson is that the project should focus considerable resources on specification language design, which still contains many open research problems. Another important lesson is that, to support such a specification language, the project needs to involve groups doing research on extensible compilers and integrated development environments.

1 Introduction

Hoare's verifying compiler grand challenge is "the construction and application of a verifying compiler that guarantees correctness of a program before running it" [18, p. 63]. This challenge is of such a broad scope that a project meeting the challenge would involve "a significant section of the research community" that would "work together towards a common goal, agreed to be valuable and achievable by a team effort" [18, p. 63].

This position paper focuses on one way that these researchers could work together, drawing lessons from the experience of the Java Modeling Language (JML) project [8,19]. These lessons are relevant because the JML project, although smaller and less ambitious, has many parallels with the verifying compiler grand challenge.

2 What Kind of Specifications?

To verify a program, one must have a specification. The specification can be implicit, such as that the program should not encounter unexpected exceptions due to obvious program errors (such as dereferencing a null pointer or indexing an array beyond its bounds). But finding such bugs is possible with existing tools, such as ESC/Java [14]. While such tools are a subject of current research and engineering, they hardly constitute a grand challenge at this point. Therefore, implicit in the grand challenge are interesting specifications, written in some

** Current address for Curtis Clifton: Dept. of Comp. Sci. and Soft. Eng., Rose-Hulman Inst. of Technology, 5500 Wabash Ave., Terre Haute, IN 47803.

specification language. Examples of interesting specifications include specifications of functional behavior that involves data values and specifications of safety properties that describe synchronization of concurrent threads.

The main cost of such interesting specifications is that, in general, they cannot be automatically generated. Instead, they must be written, to some extent, by humans. The reason human input is needed is that only humans can judge the intent of a specification. For example, consider a square root routine. A particular implementation may produce roots with 7 decimal digits of accuracy, but only a human can decide if the intended behavior is 7 digit accuracy, rather than 5 or 10 digit accuracy. Put another way, interesting specifications describe the set of all acceptable implementations abstractly; the intent behind this description allows them to be used as contracts that govern future evolution of both implementations and clients [24]. Ultimately these contracts are a matter for human judgment and negotiation.

3 Why Design a Specification Language?

Strictly speaking, the grand challenge can proceed with only a few interesting specifications. For example, a group of experts might specify the Linux kernel or write a few other interesting specifications as tests of verification technology. If the focus of the project is solely on proving programs correct, then a small set of such specifications would be adequate, and the costs of writing the specifications and designing specification languages could be largely ignored by the project.

However, we believe that including specification language design is necessary to maximize the project's impact, and would also have several other benefits.

The relationship between the project's potential impact and including specification language design in its scope can be seen by considering the alternative. Suppose that the project only works with a small set of test specifications, and does not provide an easily-usable, well-documented specification language. Assuming that the project succeeds, then how can programmers apply its verification technology to code that does not implement the test specifications? In this scenario, such applications might still be very costly, as programmers would have to write new interesting specifications, which we are assuming would be hard. Hence many programming projects would not be able to cost-effectively use the new verification technology.

Another benefit, not to be overlooked, is that a specification language is a good way to coordinate efforts among different verification tools. This is one of the main lessons of the JML project, which has been fairly successful in coordinating the efforts of diverse research groups [8]. Having JML as a common specification language allows these groups to also share users, and thus have a larger pool of users to test their ideas and to obtain feedback. It also facilitates the exchange of ideas among research groups.

Therefore, we believe that one of the project's overall goals should be to make it easy (inexpensive) and valuable (cost-effective) for ordinary programmers to write interesting formal specifications. One milestone would be to replace most informal documentation with formal specifications, while decreasing the overall cost of program development and maintenance, since at that point formal specifications become economically attractive.

Achieving such a goal requires efforts in education, language design, and tools. The educational effort needs to address documentation and training issues. This implies that documentation of the specification language, including examples, should have a high priority. It also implies that tutorial and teaching materials should have a high priority. This also presents an opportunity include in the project people interested in computer science education, with the goal of integrating more formal methods training into the standard computer science curriculum. One promising approach to doing this would be to promote undergraduate textbooks that use formal specifications.

The specification language design and tool-building efforts should have as their overall goal making it as easy as possible for programmers to read and write interesting specifications, in order to minimize educational costs. In the rest of this position paper, we will focus on these language design and tool building problems, since they are the ones we have the most experience with in JML.

4 Nature of the Specification Language

Assuming that the project will devote some effort to specification language design, we now consider what kinds of specification languages would be suitable, and whether there should be a single specification language.

A basic decision is whether the specification languages should be tailored to some specific programming language. That is, should they be interface specification languages (like Eiffel, Larch/C++, or JML), or should they be languages that are independent of any particular programming language (as are VDM, Z, or OCL)? Perhaps the specification language could even be that of some theorem prover, such as PVS or Isabelle?

We believe that the specification languages should be interface specification languages. The great advantage of an interface specification language is that it can specify details particular to some programming model, such as exceptions, visibility restrictions, encapsulation, and typing. Furthermore, an interface specification language can be translated into the input of various theorem provers and other tools, serving as a common front end for them. From the perspective of the verification tools, it may be possible to achieve some of the benefits of an interface specification language, while being somewhat language independent, by targeting an execution platform, such as the Java Virtual Machine or Microsoft's Common Language Runtime. However, even if the language targeted such an execution platform, to have impact on programmers the project would still require user-level specification languages that map to such platform-level languages.

One of the lessons of the JML project is that focusing on just a single specification language allows that language to serve as a central coordination mechanism for diverse groups of researchers. The common language:

– gives these groups a *lingua franca* in which to present semantic issues and tool-building issues,
– provides fresh insights as various technologies can be compared and contrasted through how they deal with common language features, and
– relieves groups of some of the tedium of fine-grained language design, which frees them to innovate in specialized areas.

Therefore, for the remainder of this paper, we will assume a single specification language targeting a specific programming language. Most of the arguments we raise will also apply to a lower-level specification language that targets an execution platform.

5 Problems in Specification Language Design

A major lesson of the JML project is that, even for sequential programs, the design of interface specification languages is still an interesting and quite difficult research problem.

The overall problem for specification language design is how the language can give its users sufficient value to justify the cost of specification. This is hard because, in our experience, the costs of writing a fairly complete functional specification of program behavior is usually about the same as that of writing the code to implement it. Therefore, it is necessary to either reduce these costs or to provide a wealth of tool support to compensate.

JML's approach has been to provide a wealth of tool support. This tool support works to both decrease the cost of writing specifications (for example, by improving the reporting of errors in specifications) and increase the usefulness of specifications (by using them to derive testing oracles and to generate documentation, among other things).

In addition to this tool support, JML is also designed to be easy for Java programmers to adopt and use. The main technical ideas here are to use an extended subset of Java expressions for assertions (drawing on the Eiffel experience), provide mathematical models as Java classes (drawing on the Larch and VDM experience), and prohibit side effects in expressions used in assertions [19].

Techniques for avoiding side effects in JML assertions are rather draconian at the moment and need refinement in order to be practical. The basic problem is that in a language like JML, one must be able to call methods in specifications. For example, to specify Java collections, one must be able to call the `equals` method on `Object`s. However, because of the specification inheritance used in JML to guarantee behavioral subtyping [11,20], saying that `Object`'s `equals` method is free of side effects means that no side effects are allowed in any overriding `equals` method [19]. We are working on allowing side effects that are not observable, which would allow caches in such methods, based on work by Barnett, *et al.* [3,27].

Another area of ongoing research is how to specify frame axioms (modifies clauses) and invariants in a modular fashion. To achieve modularity for frame axioms, there are several current approaches. The Boogie method [2,21] uses new language statements (`unpack` and `pack`), a dynamically unique object owner, and explicit specification of what objects are threatened by a method. While this makes soundness clear and is very flexible, it is (at present) fairly tedious to use in practice. Like the Boogie method, other techniques [22,25] seem to involve some control of aliasing. The same seems true of a modular treatment of invariants. Again, the Boogie method is more explicit, and also uses unique ownership, and other methods [26] also rely on control of aliasing. In any case some integration with alias control is necessary to prevent representation exposure. Thus these problems and current solutions seem to lead the research into the realm of alias-controlling type systems [4,28]. However, such alias-controlling type systems and their integration with specification languages are still a very active area of research. The work on JML cited above [25,26] builds on one such research project, the Universe type system [12].

Another notable area of research is how to specify and verify callbacks. Higher-order features, and especially callbacks, are well known to cause difficulties both for practical programming [30]. In specification, the main problem is that the specifications become highly parameterized [13,15], and hence difficult to write and read. In JML we are pursuing the “grey-box approach” [6,7,5]: writing specifications in a refinement-calculus style as abstract programs, but interpreting the internal callbacks as observable.

All the above issues in specification language design apply to specification languages for sequential programs. Another host of issues enters when one tries to specify concurrent programs. The JML effort has mostly ignored these problems, because it has focused on a sequential subset of Java. However, one problem that is important for concurrent programs is also a problem for sequential programs: how to specify orderings of events. That is, the ability to specify permitted sequences of operations would be a useful adjunct to traditional Hoare-style specification languages. Some specification languages have used finite-state machine descriptions for such sequencing [1]. Temporal logic is a widely-known formalism for such sequence specifications [23], although other formalisms may be easier for ordinary programmers to write and read [10,17]. There has been some preliminary work in JML on this [9].

Beyond these issues in functional specification languages is the largely unexplored territory of specification languages that combine features for specification of both functional behavior (including data values) and concurrency control (including safety and liveness). Most specification languages and tools address only one of these areas. However, modern software requires a specification language that can help integrate techniques from these two disparate camps. A start towards this problem in JML is found in the work of Rodríguez *et al.* [29]. This work uses atomicity to reduce reasoning about concurrent programs to the sequential case.

6 Keeping up with Evolution

As we argued above, to have maximal impact on practice, a specification language and its tools must be designed for practicing programmers. The problem is that widely-used languages and their programming environments evolve rapidly, and it is difficult for researchers to keep up with industrial evolution of languages and development environments. This is the case for any "standardized" language, such as C (since the standards committees issue a new standard roughly every 5 years), and perhaps more so for languages that are not standardized, such as Java.

The history of the JML project provides several lessons related to this problem. The JML project started in 1998 as an outgrowth of work on the interface specification language Larch/C++.[1] Larch/C++ was an interface specification language tailored to the specification of C++ modules. C++ was originally chosen because it was a popular object-oriented language, and we believed that we could have maximum impact by working with real problems in a broadly-used language.

However, the size and complexity of C++ presented great problems for our tool building efforts. In particular, the grammatical complexity of C++ presented enormous difficulty. In practice, another large difficulty was the unsafe nature of C++, which complicated specifications in many practical ways.

Thus when Java became available, we abandoned work on Larch/C++ and started work on an interface specification language tailored to Java—JML. Java was (at the time) a much simpler language than C++, which initially seemed to solve many of the tool problems.

Unfortunately, Java has since grown much larger than it was originally; furthermore, it has grown rather quickly. Indeed, the latest release of Java (version 1.5 also known as Java 5) introduces several non-trivial features, so it is challenging for an open-source project like JML to keep up. This dilemma is likely to be faced by the verifying compiler challenge also. In short: (a) in order to have impact, it is helpful to target a popular language, (b) but one way that a language stays popular is by evolving rapidly. This puts great pressure on fundamental tool support: parsing and basic compiler infrastructure have to be frequently updated to track the language's evolution.

Another issue for practical adoption is building integrated development environments (IDEs). Few modern programmers use the traditional command line compilers and old-style text editors; instead they demand integrated support for editing, compiling, and debugging (to say nothing of version control, support for refactoring, etc.). Thus, to have an impact—indeed, just to have users—tools must fit into some IDE. Like languages, IDEs are also evolving rapidly,

[1] The Larch family of interface specification languages [16], was a refinement of Hoare-style specification languages exemplified by VDM.

as evidenced by Eclipse[2]. Thus, while it is not a grand challenge to produce a state-of-the-art IDE, the project must keep up with their evolution.

The lesson we draw from this is that the project must have groups that are interested in building and maintaining basic compiler tool and IDE support. Researchers in formal methods are not interested in building basic compiler tools and IDEs; so the project must find people who are interested in these issues, in order to help the formal methods researchers stay focused. That is, formal methods researchers need help from some groups that are dedicated to providing up-to-date and extensible compiler support. These groups could track changes in programming language(s) and relieve some of the pressure in dealing with the evolution of popular languages. Similarly, formal methods researchers need help from groups that are dedicated to providing extensible IDEs.

The need to interact with researchers in extensible compilers and IDEs should be seen as an opportunity to involve more areas of computing in the grand challenge. One way to convince researchers interested in compilers to work with the grand challenge is to let them see additional opportunities for optimization and for exploring language extension techniques. Extensibility is needed not just to track evolution in the programming language, but also to allow experimentation in specification language design. Similarly, IDE researchers might be attracted to the grand challenge if they see opportunities to experiment with extensibility, or to improve the programmer's experience through improved error reporting, specification-enabled visualization, or editing mechanisms.

It is a fortunate fact that, at least in the United States, compiler research and formal methods research are housed in the same division of the National Science Foundation. It will be easier to get support for a grand challenge if we can find ways to involve compilers and other subareas of computing (such as education and human-computer interfaces).

7 Conclusions

To summarize, the grand challenge needs interesting specifications as tests of its verification technology. To have maximum impact, the grand challenge should have as a goal making it easy and cost-effective to write such interesting specifications. This necessarily involves some effort in specification language design. We listed some of the research challenges in specification language design from our experience with the JML project; the grand challenge should not assume that specification languages are completely understood and adequate.

The lessons from the JML project also indicate that the verifying compiler grand challenge should involve researchers in extensible compiler technologies and IDEs. Work in all these areas is necessary for the project to have a practical impact worthy of being a "grand challenge."

[2] Details on Eclipse's rapid evolution are available from `http://www.eclipse.org`.

Acknowledgments

Thanks to the program committee for comments that helped clarify our arguments.

The work of Leavens and Clifton was supported in part by the US National Science Foundation through grants CCF-0428078 and CCF-0429567.

References

1. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, Jan. 16–18, 2002.
2. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6), 2004.
3. M. Barnett, D. A. N. W. Schulte, and Q. Sun. 99.44% pure: Useful abstractions in specification. Obtained from the web at the following URL: `http://guinness.cs.stevens-tech.edu/~naumann/publications/purityJoT.pdf`, January 2005.
4. J. Boyland, J. Noble, and W. Retert. Capabilities for sharing. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming: 15th European Conference, Budapest, Hungary*, volume 2072 of *Lecture Notes in Computer Science*, pages 1–27, Berlin, June 2001. Springer-Verlag.
5. M. Büchi. Safe language mechanisms for modularization and concurrency. Technical Report TUCS Dissertations No. 28, Turku Center for Computer Science, May 2000.
6. M. Büchi and W. Weck. A plea for grey-box components. Technical Report 122, Turku Center for Computer Science, Presented at the Workshop on Foundations of Component-Based Systems, Zürich, September 1997, 1997. http://www.abo.fi/˜mbuechi/publications/GreyBoxes.html.
7. M. Büchi and W. Weck. The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science, Aug. 1999. http://www.abo.fi/˜mbuechi/publications/TR297.html.
8. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.
9. Y. Cheon and A. Perumendla. Specifying and checking method call sequences in JML. In H. R. Arabnia and H. Reza, editors, *Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP '05), Volume II, Las Vegas, Nevada, June 27-29, 2005*, pages 511–516. CSREA Press, 2005.
10. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, New York, NY, June 2000. ACM Press.
11. K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, Mar. 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

12. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 2005. To appear.
13. G. W. Ernst, J. K. Navlakha, and W. F. Ogden. Verification of programs with procedure-type parameters. *Acta Informatica*, 18(2):149–169, Nov. 1982.
14. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37, 5 of *SIGPLAN*, pages 234–245, New York, June 17–19 2002. ACM Press.
15. J. A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, Sept. 1984.
16. J. V. Guttag, J. J. Horning, S. Garland, K. Jones, A. Modet, and J. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, NY, 1993.
17. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
18. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, Jan. 2003.
19. 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*, 55(1-3):185–208, Mar. 2005.
20. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, and J. Kiniry. JML reference manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, July 2005.
21. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.
22. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Trans. Prog. Lang. Syst.*, 24(5):491–553, Sept. 2002.
23. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, New York, NY, 1992.
24. B. Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, NY, second edition, 1997.
25. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency, Computation Practice and Experience.*, 15:117–154, 2003.
26. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. Technical Report 424, ETH Zurich, Mar. 2005.
27. D. A. Naumann. Observational purity and encapsulation. In *Fundamental Aspects of Software Engineering (FASE)*, 2005. Obtained from the author.
28. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer-Verlag, July 1998.
29. E. Rodríguez, M. B. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. Technical Report SAnToS-TR2004-10, Kansas State University, Department of Computing and Information Sciences, May 2005. To appear in *ECOOP 2005.*

30. C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, second edition edition, 2002.