JML: a Java Modeling Language

Gary T. Leavens[*], Albert L. Baker, and Clyde Ruby
Department of Computer Science, 226 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1040 USA
leavens@cs.iastate.edu, baker@cs.istate.edu, ruby@cs.iastate.edu

September 18, 1998

Abstract

JML is a behavioral interface specification language tailored to Java. It also allows assertions to be intermixed with Java code, as an aid to verification and debugging. JML is designed to be used by working software engineers, and requires only modest mathematical training. To achieve this goal, JML uses Eiffel-style assertion syntax combined with the model-based approach to specifications typified by VDM and Larch. However, JML supports quantifiers, specification-only variables, frame conditions, and other enhancements that make it more expressive for specification than Eiffel.

This paper discusses the goals of JML, the overall approach, and prospects for giving JML a formal semantics through a verification logic.

1 Introduction

JML [23], which stands for "Java Modeling Language," is a behavioral interface specification language (BISL) [44] designed to specify Java [2, 9] modules. Java *modules* are classes and interfaces. A *behavioral interface specification* describes both the interface details of a module, and its behavior. The interface details are written in the syntax of the programming language; thus JML uses Java declaration syntax. In JML behavioral specifications are written using pre- and postconditions.

1.1 Goals

The long-term goal of our research is to better understand how to develop BISLs (and BISL tools) that are practical and effective. We are concerned with both technical requirements and with other factors such as training and documentation, although in the rest of this paper we will only be concerned with technical requirements. The practicality and effectiveness of JML will be judged by how well it can document reusable class libraries, frameworks, and Application Programmer Interfaces (APIs).

We believe that to meet the overall goal of practical and effective behavioral interface specification, JML must meet the following subsidiary goals.

[*]The work of Leavens and Ruby is supported in part by a grant from Rockwell International Corporation and by NSF grant CCR-9503168. The work of Leavens and Baker is supported in part by the NSF grant CCR 9803843.

- JML must be able to document the interfaces and behavior of existing software, regardless of the analysis and design methods used to create it.

  If JML were limited to only handling certain Java features or certain kinds of software, then some APIs would not be amenable to documentation using JML. Since the effort put into writing such documentation will have a proportionally larger payoff for software that is more widely reused, it is important to be able to document existing reusable software components. This is especially true since software that is implemented and debugged is more likely to be reused than software that has yet to be implemented.

- The notation used in JML should be readily understandable by Java programmers, including those with only standard mathematical training.

  A preliminary study by Finney [8] indicates that graphical mathematical notations, such as those found in Z [11, 37] may make such specifications hard to read, even for programmers trained in the notation. This accords with our experience in teaching formal specification notations to programmers. Hence, our strategy for meeting this goal has been to shun most special-purpose mathematical notations in favor of Java's own expression syntax.

- The language must be capable of being given a rigorous, formal semantics, and must also be amenable to tool support.

  This goal also helps ensure that the specification language does not suffer from logical problems, which would make it less useful for static analysis, prototyping, and testing tools.

We also have in mind a long range goal of a specification compiler, that would produce prototypes from constructive specifications [42]. Intermediate steps toward this goal would include an assertion checker, which would simply evaluate constructive assertions, and a prototyping tool for methods, which would construct the post-state values for a method for a given pre-state and list of actuals.

As a general strategy for achieving these goals, we have tried to blend the Eiffel [31, 32, 33] and Larch [10, 21, 44, 45] approaches to specification. From Eiffel we have taken the idea that assertions can be written in a language that is based on Java expressions. We also use the `old` notation from Eiffel, as described below, instead of the Larch style annotation of names with state functions. However, Eiffel specifications, as written by Meyer, are typically not as complete as model-based specifications written, for example, in Larch BISLs or VDM [13]. For example, Meyer partially specifies a `remove` (i.e., pop) operation for stacks as requiring that the stack not be empty, and ensuring that the stack value in the post-state has one fewer items than in the pre-state [33, p. 339]. However, the only characterization of which item is removed is given informally as a comment. To allow more complete specifications, we need ideas from model-based specification languages.

The semantic differences from Eiffel (and its cousins Sather and Sather-K) allow one to write specifications as in model-based specification languages. The most important of these is JML's use of specification-only declarations. These `model` declarations, as will be explained below, allow more abstract and exact specifications of behavior than is typically done in Eiffel; they allow one to write specifications that are similar to the spirit of VDM or Larch BISLs. A major difference is that we have extended the syntax of Java expressions with quantifiers and other constructs that are needed for logical expressiveness, but which are not always executable. Finally, we ban side-effects and other problematic features of code in assertions.

On the other hand, our experience with Larch/C++ has taught us to adapt the model-based approach in two ways, with the aim of making it more practical and easy to learn. The first adaptation is again the use of specification-only model (or ghost) variables. An object will thus have (in general) several such *model fields*, which are used only for the purpose of describing, abstractly, the values of objects. This simplifies the use of JML, as compared with most Larch BISLs, since specifiers (and their readers) hardly ever need to know about algebraic style specification. It also makes designing a model for a Java class or interface similar, in some respects, to designing an implementation data structure in Java. We hope that this similarity will make the specification language easier to understand. (This kind of model also has some technical advantages that will be described below.)

The second adaptation is hiding of the details of mathematical modeling behind a facade of Java classes. In the Larch approach to behavioral interface specification [44], the mathematical notation used in assertions is presented directly to the specifier. This allows the same mathematical notation to be used in many different specification languages. However, it also means that the user of such a specification language has to learn a notation for assertions that is different than their programming language's notation for expressions. In JML we use a compromise approach, hiding these details behind Java classes. These classes are pure, in the sense that they reflect the underlying mathematics, and hence do not use side-effects (at least not in any observable way). Besides insulating the user of JML from the details of the mathematical notation, this compromise approach also insulates the design of JML from the details of the mathematical logic used for theorem proving.

1.2 Outline

Section 2 uses examples to show how Java classes and interfaces are specified in JML. Section 3 discusses prospects for defining the semantics of JML formally, using a verification logic. Section 4 presents conclusions.

2 Class and Interface Specifications

In this section we give an example of a JML class specification and describe some of the features of JML. (These features can also be used to provide specifications for Java interfaces.)

2.1 Abstract Models

A simple example of an abstract class specification is the ever-popular `UnboundedStack` type, which is presented in Figure 1. This figure has the abstract values of stack objects specified by the model data field `theStack`, which is declared on the fourth non-blank line. Since it is declared using the modifier `model`, such a field does not have to be implemented; however, for purposes of the specification we treat it exactly as any other Java field (i.e., as a variable). That is, we imagine that each instance of the class `UnboundedStack` has such a field.

The type of the model field `theStack` is a pure type, `JMLObjectSequence`, which is a sequence of objects. It is provided by JML in the package `edu.iastate.cs.jml.models`,

```
package edu.cs.iastate.jml.samples.stacks;

//@ model import edu.cs.iastate.jml.models.*;

public abstract class UnboundedStack {

  //@ public model JMLObjectSequence theStack;

  //@ public initially theStack.isEmpty();

  public abstract void pop( );
    //@ behavior {
    //@   requires !theStack.isEmpty();
    //@   modifiable theStack;
    //@   ensures theStack.equals(old(theStack.trailer()));
    //@ }

  public abstract void push(Object x);
    //@ behavior {
    //@   modifiable theStack;
    //@   ensures theStack.equals(old(theStack.addFirst(x)));
    //@ }

  public abstract Object top( );
    //@ behavior {
    //@   requires !theStack.isEmpty();
    //@   ensures result == theStack.first();
    //@ }
}
```

Figure 1: A specification of the abstract class `UnboundedStack` (file `UnboundedStack.java`).

which is imported in the second non-blank line of the figure.[1] Note that this `import` declaration does not have to appear in the implementation, since it is modified by the keyword `model`. In general, any declaration form in Java can have this modifier, with the same meaning: that the declaration in question is only used for specification purposes, and does not have to appear in an implementation.

Following the declaration of the model field, above the specification of `pop` in Figure 1, is an `initially` clause. (Such clauses are adapted from Resolve [34].) This clause is declared `public`, since it only refers to public model fields.

An `initially` clause permits data type induction [12, 46] for abstract classes and interfaces, by supplying a property that must appear to be true of the starting states of objects. In each visible state (outside of the methods of `UnboundedStack`) all reachable objects of the type `UnboundedStack` must have a value that makes them appear to have been created as empty stacks and subsequently modified using the type's methods.

Following the `initially` clauses are the expected specifications of the `pop`, `push`, and `top` methods.

The use of the `modifiable` clauses in the behavioral specifications of `pop` and `push` is interesting (and another difference from Eiffel). These give frame conditions [4], which say that no objects, other than those mentioned (and those on which these objects depend, as explained below) may have their values changed.[2] When the `modifiable` clause is omitted, as it is in the specification of `top`, this means that no objects can have their state modified by the method's execution. Our interpretation of this is very strict, as even benevolent side effects are disallowed if the `modifiable` clause is omitted [27, 26].

When a method can modify some objects, these objects have different values in the pre-state and post-state of that method. Often the post-condition must refer to both of them. A notation similar to Eiffel's is used to refer to the pre-state value of a variable. In JML the syntax is `old(`E`)`.[3] The meaning of `old(`E`)` is as if E were evaluated in the pre-state and that value is used in place of `old(`E`)` in the assertion. This is sensible if E denotes a primitive value (such as an `int`), or if the type of E is a pure type. If E denotes an object that is modifiable, then the expression may not mean what is desired. For example, if `a` is a Java array, then

```
old(a)[3] == a[3]
```

does not constrain the value of `a`'s third element, because it only saves a reference to `a`, while the following does,

```
old(a[3]) == a[3]
```

because it saves the value of `a[3]`.

For example, in `pop`'s postcondition the expression `old(theStack.trailer())` has type `JMLObjectSequence`, which is a pure type. The value of `theStack.trailer()` is computed in the *pre-state* of the method (just after the method is called and parameters have been passed, but before execution of the body).

[1] Users can also define their own pure types, as we will explain below.

[2] An object is modified by a method when it is allocated in both the pre- and post-states of the method, and when some of its variables (model or concrete) change their values. This means that allocating objects, using Java's `new` operator, does not cause a modification.

[3] We use explicit parentheses following `old`, which indicates the expression to be evaluated in the pre-state explicitly; this is a difference from Eiffel.

Note also that, since `JMLObjectSequence` is a reference type, one is required to use `equals` instead of `==` to compare them for equality of values. (Using `==` would be a mistake, since it would only compare them for object identity, which in combination with `new` would always yield false.)

The specification of `push` does not have a `requires` clause. This means that the method imposes no obligations on the caller. (Logically, the meaning of an omitted `requires` clause is that the method's precondition is `true`, which is satisfied by all states, and hence imposes no obligations on the caller.) This seems to imply that the implementation must provide a literally unbounded stack, which is surely impossible. We avoid this problem, by following Poetzsch-Heffter [35] in releasing implementations from their obligations to fulfill the postcondition when Java runs out of storage. That is, a method implementation is correct if, whenever it is called in a state that satisfies its precondition, either

- the method terminates in a state that satisfies its postcondition, having modified only the objects permitted by its `modifiable` clause, or
- Java signals an error, by throwing an exception that inherits from `Error`.

2.2 Other Aspects of JML

While the example in this short paper does not require its use, JML does support quantified assertions. The following simple examples illustrate JML quantified assertions:

```
forall (int i) [intSet.isIn(i) => i > 0]
  // all elements of intSet are positive
exists(int i) [intSet.isIn(i) && i % 2 == 0]
  // there is an even element of intSet
```

Following Leino [27, 26], JML uses `depends` and `represents` clauses to relate model fields to the concrete fields of objects. A `depends` clause, such as the following,

```
depends size on theElems;
```

says that the model field `size` may change its value when `theElems` changes. A `represents` clause says how they are related, giving additional facts that can be used in reasoning about the specification. This serves the same purpose as an abstraction function in various proof methods for abstract data types (such as [12]). For example,

```
represents size by size == theElems.length();
```

tells how to extract the value of `size` from the value of `theElems`.

JML also has invariants and history constraints [29]. A history constraint is used to say how values can change between earlier and later states, such as a method's pre-state and its post-state. This prohibits subtypes from making certain state changes, even if they implement more methods than are specified in a given class. For example, the following history constraint

```
constraint MAX_VALUE == old(MAX_VALUE);
```

says that the value of `MAX_SIZE` cannot change.

JML has the ability to specify what methods a method may call, using a `callable` clause. This allows one to know which methods need to be looked at when overriding a method [14], and to apply the ideas of "reuse contracts" [38].

JML also features checkable redundancy [22, 39, 40]. In JML this is usually signaled by the keyword `redundantly`. For example, one can write a redundant invariant, history constraint, precondition or postcondition by writing `invariant redundantly`, `constraint redundantly`, `requires redundantly`, or `ensures redundantly`. Such clauses state that the property is believed to follow from the other properties of the specification. For example, a redundant invariant should follow from other stated invariants. Another kind of checkable redundancy is an `example` clause [19, 22], which can be used to give concrete examples of a method's execution. Such redundancy can be used as a rhetorical device, to bring various properties to the attention of the specification's readers.

Following Wing and Wills [46, 43], a specification may be written using several cases separated by the keyword `also` [20]. The semantics is that, when the precondition of a case is satisfied, the rest of that case's specification must be obeyed. Separating the specification into several cases is useful in specifying operations that may signal exceptions,a and for giving an interpretation of behavioral subtyping [7].

In JML, a subtype inherits the specifications of its supertype's public and protected members (fields and methods), as well as invariants and history constraints as additional specification cases. This ensures that a subclass specifies a behavioral subtype of its supertypes.

2.3 Making New Pure Types

JML comes with a suite of pure types, implemented as Java classes. At the time of this writing these are `JMLObjectSet`, `JMLObjectSequence`, `JMLObjectMap` and `JMLValueSet`, `JMLValueSequence`, `JMLValueMap`, `JMLInteger`, and a few helper classes (such as exceptions and enumerators). These are found in the package `edu.iastate.cs.jml.models`, and can be used for defining abstract models. Users can also create their own pure types if desired. Since these types are to be treated as purely immutable values in specifications, they must pass certain conservative checks that make sure there is no possibility of observable side-effects from using such objects.

Model classes should also be pure, since there is no way to use non-pure operations in an assertion. However, the modifiers `model` and `pure` are orthogonal, and thus usually one will have to list both of them when declaring a model class.

3 Prospects for Verification

We plan to design a verification logic for Java programs that will help us:

- define the semantics of JML specifications precisely, by giving the verification conditions for method specifications,
- support checking for errors in Java programs (as in SRC's extended static checker project),
- correctness proofs, and
- allow other kinds of formal analysis of the properties of Java programs.

There are several problems we anticipate in designing a verification logic for Java programs that can be used with JML. These include:

- termination,
- side-effects in expressions,
- aliasing,
- subtyping and dynamic dispatch.

To deal with termination, we adapt the well-known idea of giving a separate proof of termination, using variant functions. (We have adopted Resolve's [34] syntax for declaring variant functions in while loops for this.)

To deal with side-effects in expressions, we plan to restructure the code to be verified into simpler statements [3, 15]. These statements would not include any compound expressions. For example, the Java statement:

```
v = o.f(x++);
```

would be treated as the following, for purposes of verification (assuming that `x` and `v` are both of type `int`).

```
int temp1 = x;
x = x + 1;
v = o.f(temp1);
```

To deal with aliasing, we plan to require that the verifier verify a method for each possible case of aliases among the names it uses. Often some of these cases can be ruled out by preconditions.

To deal with subtyping and dynamic dispatch, we plan to use behavioral subtyping and supertype abstraction [1, 5, 17, 18, 24, 25, 29, 30, 41]. Since JML forces subtypes to be behavioral subtypes [6], this allows one to reason about Java programs using the static types of variables and expressions, ignoring dynamic dispatch.

4 Future Work and Conclusions

One area of future work for JML is concurrency. Our current plan is to use `when` clauses that say when a method may proceed to execute, after it is called [28, 36]. This permits the specification of when the caller is delayed to obtain a lock, for example. While syntax for this exists in the JML parser, our exploration of this topic is still in an early stage. We may also be able to expand history constraints to use temporal logic.

Another fertile area for future work on JML is to synthesize the previous work of Wahls, Leavens and Baker on the use of constraint logic programming to directly execute a significant and practical subset of JML's assertions [42]. This prior work supports the "construction" of post-state values to satisfy ensures clauses, including such clauses containing quantified assertions. Successful integration of these assertion execution techniques with JML would support automatic generation of Java class prototypes directly from their JML specifications.

JML combines the best features of Eiffel and the Larch approaches to specification. This combination, we believe, makes it more expressive than Eiffel, and more practical than Larch style BISLs. We look forward to precisely describing the semantics of JML using a verification logic.

Acknowledgements

Thanks to Rustan Leino and Peter Müller for many discussions about the semantics of such specifications and verification issues relating to Java. For comments on JML we thank Peter, Jianbing Chen, Anand Ganapathy, Sevtap Oltes, Gary Daugherty, Karl Hoech, Jim Potts, and Tammy Scherbring.

References

[1] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bezivin et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 234–242, New York, N.Y., June 1987. Springer-Verlag. Lecture Notes in Computer Science, Volume 276.

[2] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.

[3] J. W. De Bakker, J. W. Klop, and J.-J. Ch. Meyer. Correctness of programs with function procedures. In D. Kozen, editor, *Logics of Programs*, number 131 in Lecture Notes in Computer Science, pages 94–112. Springer-Verlag, New York, N.Y., 1982.

[4] Alex Borgida, John Mylopoulos, and Rayomnd Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[5] Krishna Kishore Dhara. Behavioral subtyping in object-oriented languages. Technical Report TR97-09, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames IA 50011-1040, May 1997. The author's Ph.D. dissertation.

[6] Krishna Kishore Dhara and Gary 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, March 1996.

[7] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. Technical Report 95-20c, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 1997. Also in Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, 1996, pp. 258–267. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[8] Kate Finney. Mathematical notation in formal specification: Too difficult for the masses? *IEEE Transactions on Software Engineering*, 22(2):158–159, February 1996.

[9] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, 1996.

[10] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, N.Y., 1993.

[11] I. Hayes, editor. *Specification Case Studies*. International Series in Computer Science. Prentice-Hall, Inc., second edition, 1993.

[12] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[13] Cliff B. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[14] Gregor Kiczales and John Lamping. Issues in the design and documentation of class libraries. *ACM SIGPLAN Notices*, 27(10):435–451, October 1992. *OOPSLA '92 Proceedings*, Andreas Paepcke (editor).

[15] H. Langmaack. Aspects of programs with finite modes. In M. Karpinski, editor, *Foundations of Computation Theory*, number 158 in Lecture Notes in Computer Science, pages 241–254. Springer-Verlag, New York, N.Y., 1983.

[16] K. Lano and H. Haughton, editors. *Object-Oriented Specification Case Studies*. The Object-Oriented Series. Prentice Hall, New York, N.Y., 1994.

[17] Gary T. Leavens. Modular verification of object-oriented programs with subtypes. Technical Report 90-09, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, July 1990. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[18] Gary T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, July 1991.

[19] Gary T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In Haim Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, Boston, 1996. An extended version is TR #96-01d, Department of Computer Science, Iowa State University, Ames, Iowa, 50011.

[20] Gary T. Leavens. Larch/C++ Reference Manual. Version 5.14. Available in ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz or on the World Wide Web at the URL http://www.cs.iastate.edu/~leavens/larchc++.html, October 1997.

[21] Gary T. Leavens. Larch frequently asked questions. Version 1.89. Available in http://www.cs.iastate.edu/~leavens/larch-faq.html, January 1998.

[22] Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. Technical Report 97-19, Iowa State University, Department of Computer Science, September 1997.

[23] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06a, Iowa State University, Department of Computer Science, July 1998.

[24] Gary T. Leavens and William E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). In N. Meyrowitz, editor, *OOPSLA ECOOP '90 Proceedings*, volume 25(10) of *ACM SIGPLAN Notices*, pages 212–223. ACM, October 1990.

[25] Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, November 1995.

[26] K. Rustan M. Leino. A myth in the modular specification of programs. Technical Report KRML 63, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue Palo Alto, CA 94301, November 1995. Obtain from the author, at rustan@pa.dec.com.

[27] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[28] Richard Allen Lerner. Specifying objects of concurrent systems. Ph.D. Thesis CMU-CS-91-131, School of Computer Science, Carnegie Mellon University, May 1991.

[29] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[30] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., 1988.

[31] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[32] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.

[33] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.

[34] William F. Ogden, Murali Sitaraman, Bruce W. Weide, and Stuart H. Zweben. Part I: The RESOLVE framework and discipline — a research synopsis. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, Oct 1994.

[35] Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.

[36] Gowri Sivaprasad. Larch/CORBA: Specifying the behavior of CORBA-IDL interfaces. Technical Report 95-27a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 1995.

[37] J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, New York, N.Y., second edition, 1992.

[38] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *OOPSLA '96 Conference on Object-Oriented Programming Systems, Languagges and Applications*, pages 268–285. ACM Press, October 1996. ACM SIGPLAN Notices, Volume 31, Number 10.

[39] Yang Meng Tan. Interface language for supporting programming styles. *ACM SIGPLAN Notices*, 29(8):74–83, August 1994. Proceedings of the Workshop on Interface Definition Languages.

[40] Yang Meng Tan. *Formal Specification Techniques for Engineering Modular C Programs*, volume 1 of *Kluwer International Series in Software Engineering*. Kluwer Academic Publishers, Boston, 1995.

[41] Mark Utting and Ken Robinson. Modular reasoning in an object-oriented refinement calculus. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction, Second International Conference, Oxford, U.K., June/July*, volume 669 of *Lecture Notes in Computer Science*, pages 344–367. Springer-Verlag, New York, N.Y., 1992.

[42] Tim Wahls, Gary T. Leavens, and Albert L. Baker. Executing formal specifications with constraint programming. Technical Report 97-12a, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, August 1998. Available by anonymous ftp from ftp.cs.iastate.edu or by e-mail from almanac@cs.iastate.edu.

[43] Alan Wills. Refinement in Fresco. In Lano and Houghton [16], chapter 9, pages 184–201.

[44] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[45] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–24, September 1990.

[46] Jeannette Marie Wing. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.