JML: A Notation for Detailed Design

by Gary T. Leavens, Albert L. Baker, and Clyde Ruby

This paper is adapted from Haim Kilov, Bernhard Rumpe, and William Harvey (editors), *Behavioral Specifications for Businesses and Systems*, chapter 12, pages 175-188.

1 Behavioral Interface Specification

Abstract

JML is a behavioral interface specification language tailored to Java. It is designed to be written and read by working software engineers, and should require only modest mathematical training. It uses Eiffel-style syntax combined with model-based semantics, as in VDM and Larch. JML supports quantifiers, specification-only variables, and other enhancements that make it more expressive for specification than Eiffel and easier to use than VDM and Larch.

JML [Leavens-Baker-Ruby01], which stands for "Java Modeling Language," is a behavioral interface specification language (BISL) [Wing87] designed to specify Java [Arnold-Gosling98] [Gosling-Joy-Steele96] modules. Java *modules* are classes and interfaces.

A *behavioral interface specification* describes both the details of a module's interface with clients, and its behavior from the client's point of view. Such specifications are not good for the specification of whole programs, but are good for recording detailed design decisions or documentation of intended behavior, for a software module.

The goal of this chapter is to explain JML and the concepts behind its approach to specification. Since JML is used in detailed design of Java modules, we use the detailed design of an interface and class for priority queues as an example. The rest of this section explains interfaces and behavioral interface specification. In the next section we describe how to specify new types as conceptual models for detailed design. Following that we finish the example by giving the details of a class specification. We conclude after mentioning some other features of JML.

1.1 Interfaces

A module's *interface* consists of its name, and the names and types of its fields and methods. Java interfaces declare such interface information, but class declarations do as well. As in the Larch family of BISLs [Guttag-Horning93] [LeavensLarchFAQ] [Wing87] [Wing90a], interface information in JML is declared using the declaration syntax of the programming language to which the BISL is tailored; thus, JML uses Java declaration syntax.

An example is given in the file '`PriorityQueueUserInterface.java-refined`', which is shown below. This example gives the information a Java program needs to use a `PriorityQueueUser` object, including the package to which it belongs, the accessibility of the methods (`public`), the names of the methods, the types of their arguments and results, and what exceptions they can throw.

```
package edu.iastate.cs.jml.docs.kluwer;
public interface PriorityQueueUser {
 /*@ pure @*/ boolean contains(Object argID);
 /*@ pure @*/ Object next() throws PQException;
 void remove(Object argID);
}
```

Also included in the above file are two annotations. These annotations are enclosed within annotation comments of the form `/*@ ... @*/`; one can also write annotation comments using the form `//@`, and such comments extend to the end of the corresponding line.

Java ignores both kinds of annotation comments, but they are significant to JML. The annotations on the methods `next` and `contains` require both methods to be `pure`, meaning that they cannot have any externally-visible side effects.

1.2 A First Example of Behavioral Specification

In JML, behavioral specification information is also given in the form of annotations. As in the Larch approach, such specifications are model-based. That is, they are stated in terms of a mathematical model [Guttag-Horning93] [Hoare72a] [Wing83] [Wing87] of the states (or values) of objects. Unlike most Larch-style specification languages, however, in JML such models are described by declaring *model fields*, which are only used for purposes of specification. In JML, a declaration can include the modifier `model`, which means that the declaration need not appear in a correct implementation; all non-model declarations must appear in a correct implementation.

As an example, the file '`PriorityQueueUserModel.java-refined`' below specifies a model for priority queues. This specification is a refinement of the one given in the file (shown above) '`PriorityQueueUserInterface.java-refined`', which is why the `refine` clause appears in the specification following the `package` declaration. The meaning of the `refine` clause is that the given specification adds to the one in the file named, by imposing additional constraints on that specification. Such a refinement might be done, for example, when one is starting to make detailed design decisions or when starting to specify the behavior of existing software modules. In a refinement, existing specification information is inherited; that is, the method declarations in the interface `PriorityQueueUser` are inherited, and thus not repeated below

```
package edu.iastate.cs.jml.docs.kluwer;

//@ refine PriorityQueueUser <- "PriorityQueueUser.java-refined";
//@ model import edu.iastate.cs.jml.models.*;

public interface PriorityQueueUser {

 /*@ public model instance JMLValueSet entries
   @      initially entries != null && entries.isEmpty();
   @*/
 /*@ public invariant entries != null
   @    && (\forall JMLType e; entries.has(e);
   @                          e instanceof QueueEntry);
   @ public invariant
   @   (\forall QueueEntry e1; entries.has(e1);
   @       (\forall QueueEntry e2;
   @         entries.has(e2) && !(e1.equals(e2));
   @           e2.iD != e1.iD
   @           && e2.timeStamp != e1.timeStamp ) );
   @*/
}
```

Following the `refine` clause above is a `model import` declaration. This has the effect like a Java `import` declaration for JML, but the use of `model` means that the import does not have to appear in an implementation, as it is only needed for specification purposes. The package being imported, `edu.iastate.cs.jml.models`, consists of several pure classes including sets, sequences, relations, maps, and so on, which are useful in behavioral specification. These fill the role of the built-in types used for specification in VDM and Z, or the traits used in Larch. Since they are pure (side-effect free) classes, they can be used in assertions without affecting the state of the computation, which allows assertions to have a well-defined mathematical meaning (unlike Eiffel's assertions). However, since they are Java classes, their methods are invoked using the usual Java syntax.

In the specification above we use the class `JMLValueSet` as the type of the model field `entries`. That is, for purposes of specification, we imagine that every object that implements the interface `PriorityQueueUser` has a public field `entries` of type `JMLValueSet`. This model field appears (to clients) to have started out initially as empty, as stated in the `initially` clause attached to its declaration [Ogden-etal94] [Morgan94].

The two `invariant` clauses further describe the intended state of `entries`. The first states that it is not null, and that all of its elements have type `QueueEntry`. The `\forall` notation is an addition to the Java syntax for expressions; it gives universal quantification over the declared variables. Within such an expression of the form `(\forall T x; R(x); P(x))`, the expression `R(x)` specifies the *range* over which the bound variable, `x`, can take on values; it is separated from the *term* predicate, `P(x)`, by a semicolon (`;`). For example, the first invariant means that for all `JMLType` objects `e` such that `entries.has(e)`, `e` has type `QueueEntry`. The second invariant states that every such `QueueEntry` object has a unique `iD` and `timeStamp`.

In the file '`PriorityQueueUser.java`' below we make yet another refinement, to specify the behavior of the methods of `PriorityQueueUser`. This specification, because it refines the specification in '`PriorityQueueUserModel.java-refined`', inherits the model fields specified there, as well as the `initially` and `invariant` clauses. (Inheritance of specifications is explained further below.)

```
package edu.iastate.cs.jml.docs.kluwer;
//@ refine PriorityQueueUser <- "PriorityQueueUserModel.java-refined";
//@ model import edu.iastate.cs.jml.models.*;
public interface PriorityQueueUser {

 /*@ public normal_behavior
   @   ensures \result <==>
   @          argID != null
   @          && (\exists QueueEntry e; entries.has(e);
   @                       e.iD == argID);
   @*/
 /*@ pure @*/ boolean contains(Object argID);

 /*@   public normal_behavior
   @     requires !entries.isEmpty();
   @     ensures
   @       (\exists QueueEntry r;
   @           entries.has(r) && \result == r.iD;
   @           (\forall QueueEntry o;
   @             entries.has(o) && !(r.equals(o));
```

```
    @                r.priorityLevel >= o.priorityLevel
    @                && r.timeStamp < o.timeStamp ) );
    @ also
    @   public exceptional_behavior
    @     requires entries.isEmpty();
    @     signals (PQException);
    @*/
  /*@ pure @*/ Object next() throws PQException;

  /*@   public normal_behavior
    @     requires argID != null && contains(argID);
    @     assignable entries;
    @     ensures (\exists QueueEntry e;
    @         \old(entries.has(e)) && e.iD == argID;
    @         entries.equals(\old(entries.remove(e))));
    @ also
    @   public normal_behavior
    @     requires argID == null || !contains(argID);
    @     ensures \not_modified(entries);
    @*/
  void remove(Object argID);
}
```

The specification of `contains` above shows the simplest form of a behavioral specification for a method: a single `public normal_behavior` clause followed by a method header. This specification says that the method returns true just when its argument is non-null and is the same as some object in the queue. The `public normal_behavior` clause in this specification consists of a single `ensures` clause. This `ensures` clause gives the method's total-correctness postcondition; that is, calls to `contains` must terminate (as opposed to looping forever or aborting) in a state that satisfies the postcondition. The `public` keyword says that the specification is intended for clients; while the "normal" in `normal_behavior` prohibits throwing exceptions. The meaning of `&&` and `==` are as in Java; that is, `&&` is short-circuit logical conjunction, and `e.iD == argID` means that `e.iD` and `argID` are the same object. The keyword `\result` denotes the return value of the method, which in this case is a boolean. The operator `<==>` means "if and only if"; it is equivalent to `==` for booleans, but has a lower precedence. The notation `\exists` is used for existential quantification. Like universal quantifiers, existential quantifiers can also have a range expression that is separated from the term expression by a semicolon (`;`).

The specification of the method `next` shows one way to specify methods with exceptions in JML. This uses a `public normal_behavior` clause for the case where no exceptions are thrown, and an `public exceptional_behavior` clause for when exceptions are thrown. The semantics is that a correct implementation must satisfy both of these behaviors [Leavens-Baker99] [Wills94] [Wing83]. In the specification of `next`, the `public exceptional_behavior` clause states that an instance of the `PQException` class (not shown here) must be thrown when `entries` is empty. The `requires` clause gives a precondition for that case, and when it is true, the method must terminate (in this case by throwing an exception), as that case's postcondition must be satisfied.

The public normal behavior of `next` must be obeyed when its precondition is true; that is, when `entries` is not empty. The normal behavior's postcondition says that `next` returns an object with the lowest timestamp in the highest priority level.

It would, of course, be possible to only specify the public normal behavior for `next`. If this were done, then implementations could just assume the precondition of the normal behavior—that `entries` is not empty. That would be an appropriate design for clients that can be trusted, and might permit more efficient implementation. The given specification is appropriate for untrusted clients [Meyer92a] [Meyer97].

The specification `remove` uses case analysis [Leavens-Baker99] [Wills94] [Wing83] in the specification of normal behavior. The two cases are separated by the keyword `also`, and each must be obeyed when its precondition is true. The first case contains a `assignable` clause.[1] This is a frame condition [Borgida-Mylopoulos-Reiter95]; it states that only the fields mentioned (and any on which they depend [Leino95] [Leino95a]) can be assigned to; no other fields, including fields in other objects, can be assigned. Omitting the `assignable` clause means that no fields can be assigned. (Technically, the assignable clause is also concerned with array elements. Local variables, including the formal parameters of a method, and also fields of newly-created objects may also be freely assigned by a method [Leavens-Baker-Ruby01].) Note that the precondition of `remove` uses the method `contains`, which is permitted because it is pure.

The most interesting thing about the specification of `remove` is that it uses the JML reserved word `\old`. As in Eiffel, 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.

While we have broken up the specification of `PriorityQueueUser` into three pieces, that was done partly to demonstrate refinement and partly so that each piece would fit on a page. In common use, this specification would be written in one file.

[1] For historical reasons, JML also allows one to use `modifiable` and `modifies` as synonyms for `assignable`.

2 Specifying New Pure Model Types

JML comes with a suite of pure types, implemented as Java classes, that can be used as conceptual models in detailed design. As mentioned above, these are found in the package `edu.iastate.cs.jml.models`.

Users can also create their own pure types, by giving a class or interface the `pure` modifier. 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, in JML, the use of non-pure methods in an assertion is a type error. However, the modifiers `model` and `pure` are orthogonal, and thus one must list both of them when declaring a pure model class.

An example of a pure model class is the class `QueueEntry`, specified in the file '`QueueEntry.jml`' below. Since it is a model class, it need not be implemented, but is used only for specification purposes. Since it is pure, none of its methods can permit side-effects. It is written in a '`.jml`' file, and that filename suffix tells JML that it consists solely of annotations; in effect, the entire file is ignored by Java but is significant to JML. The class `QueueEntry` has three public fields `iD`, `priorityLevel`, and `timeStamp`. The invariant clause states that the `iD` field cannot be `null` in a client-visible state.

```
package edu.iastate.cs.jml.docs.kluwer;
//@ model import edu.iastate.cs.jml.models.JMLType;

/*@ public pure model class QueueEntry implements JMLType {
  @ public Object iD;
  @ public int priorityLevel;
  @ public int timeStamp;

  @ public invariant iD != null && timeStamp >= 0;

  @  public normal_behavior
  @    requires argID != null && timeStamp >= 0;
  @    assignable iD, priorityLevel, timeStamp;
  @    ensures iD == argID && priorityLevel == argLevel
  @          && timeStamp == argTimeStamp;
  @
  @ public QueueEntry(Object argID, int argLevel,
  @                   int argTimeStamp);

  @  public normal_behavior
  @    ensures \result instanceof QueueEntry;
  @    ensures_redundantly
  @          ((QueueEntry)\result).equals(this);
  @
  @ public Object clone();

  @ also
  @  public normal_behavior
```

```
@    requires o instanceof QueueEntry;
@    ensures \result <==>
@        ((QueueEntry)o).iD == iD
@     && ((QueueEntry)o).priorityLevel == priorityLevel
@     && ((QueueEntry)o).timeStamp == timeStamp;
@ also
@  public normal_behavior
@    requires !(o instanceof QueueEntry);
@    ensures \result == false;
@
@ public boolean equals(Object o);
@ }
@*/
```

In the above specification, the constructor's specification follows the invariant. The constructor takes three arguments and initializes the fields from them. The precondition of this constructor states that it can only be called if the `argID` argument is not null; if this were not true, then the invariant would be violated.

The `clone` and `equals` methods in `QueueEntry` are related to the interface `JMLType`, which `QueueEntry` extends. In JML when a class implements an interface, it inherits the specifications of that interface. The interface `JMLType` specifies just these two methods. The specifications of these methods are thus inherited by `QueueEntry`, and thus the specifications given here add to the given specifications. The specification of the method `clone` in `JMLType` (quoted from [Leavens-Baker-Ruby01]) is as follows.

```
/*@ also
  @   public normal_behavior
  @     ensures \result instanceof JMLType
  @          && ((JMLType)\result).equals(this);
  @*/
public /*@ pure @*/ Object clone();
```

The above specification says that, for `JMLType` objects, `clone` cannot throw exceptions, and its result must be a `JMLType` object, with the same value as `this`. (In Java, `this` names the receiver of a method call).

Inheritance of method specifications means that an implementation of `clone` must satisfy both the inherited specification from `JMLType` and the given specification in `QueueEntry`. The meaning of the method inheritance in this example is shown in below [Dhara-Leavens96]. (The modifier `pure` from the superclass can be added in here, although it is redundant for a method of a pure class.)

```
/*@ also
  @   public normal_behavior
  @     ensures \result instanceof JMLType
  @           && ((JMLType)result).equals(this);
  @ also
  @   public normal_behavior
  @     ensures \result instanceof QueueEntry;
  @     ensures_redundantly
  @             ((QueueEntry)\result).equals(this);
  @*/
public /*@ pure @*/ Object clone();
```

Satisfying both of the cases is possible because `QueueEntry` is a subtype of `JMLType`, and because JML interprets the meaning of *E1*`.equals(`*E2*`)` using the run-time class of *E1*.

The `ensures_redundantly` clause allows the specifier to state consequences of the specification that follow from its meaning [Leavens-Baker99] [Tan94] [Tan95]. In this case the predicate given follows from the inherited specification and the one given. This example shows a good use of such redundancy: to highlight important inherited properties for the reader of the (original, unexpanded) specification.

Case analysis is used again in the specification of `QueueEntry`'s `equals` method. As before, the behavior must satisfy each case of the specification. That is, when the argument `o` is an instance of type `QueueEntry`, the first case's postcondition must be satisfied, otherwise the result must be false.

3 Class Specifications

The file 'PriorityQueue.java-refined' shown below specifies PriorityQueue, a class that implements the interface PriorityQueueUser. Because this class implements an interface, it inherits specifications, and hence implementation obligations, from that interface. The specification given thus adds more obligations to those given in previous specifications.

```
package edu.iastate.cs.jml.docs.kluwer;
//@ model import edu.iastate.cs.jml.models.*;

public class PriorityQueue implements PriorityQueueUser {

 /*@ public normal_behavior
   @  assignable entries;
   @  ensures entries != null && entries.isEmpty();
   @  ensures_redundantly
   @          entries.equals(new JMLValueSet());
   @*/
 public PriorityQueue();

 /*@   public normal_behavior
   @     requires entries.isEmpty();
   @     ensures \result == 0;
   @ also
   @   public normal_behavior
   @     requires !(entries.isEmpty());
   @     ensures (\forall QueueEntry e; entries.has(e);
                          \result >= e.timeStamp);
   @ public pure model int largestTimeStamp();
   @*/

 /*@   public normal_behavior
   @     requires argID != null && !contains(argID);
   @     assignable entries;
   @     ensures entries != null
   @      && entries.equals(\old(entries.insert(
   @            new QueueEntry(argID, argPriorityLevel,
   @                           largestTimeStamp()+1))));
   @ also
   @   public exceptional_behavior
   @     requires argID == null || contains(argID);
   @     signals (PQException);
   @*/
  public void addEntry(Object argID, int argPriorityLevel)
                   throws PQException;

  public /*@ pure @*/ boolean contains(Object argID);
  public /*@ pure @*/ Object next() throws PQException;
  public void remove(Object argID);
}
```

The pure model method `largestTimeStamp` is specified purely to help make the statement of `addEntry` more comprehensible. Since it is a model method, it does not need to be implemented. Without this specification, one would need to use the quantifier found in the second case of `largestTimeStamp` within the specification of `addEntry`.

The interesting method in `PriorityQueue` is `addEntry`. One important issue is how the timestamps are handled; this is hopefully clarified by the use of `largestTimeStamp()` in the postcondition of the first specification case.

A more subtle issue concerns finiteness. Since the precondition of `addEntry`'s first case does not limit the number of entries that can be added, the specification seems to imply that the implementation must provide a literally unbounded priority queue, which is surely impossible. We avoid this problem, by following Poetzsch-Heffter [Poetzsch-Heffter97] in releasing implementations from their obligations to fulfill the postcondition when Java runs out of storage. That is, a method implementation correctly implements a specification case if, whenever it is called in a state that satisfies its precondition, either

- the method terminates in a state that satisfies its postcondition, having assigned only the locations permitted by its `assignable` clause, or
- Java signals an error, by throwing an exception that inherits from `java.lang.Error`.

4 Other Features of JML

Following Leino [Leino95] [Leino95a], JML uses `depends` and `represents` clauses to relate model fields to the concrete fields of objects. For example, in the following

```
depends size <- theElems;
represents size <- size == theElems.length();
```

the `depends` clause says that the model field `size` may change its value when `theElems` changes. The `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 [Hoare72a]). The `represents` clause above tells how to extract the value of `size` from the value of `theElems`.

JML also has history constraints [Liskov-Wing94]. 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_SIZE == \old(MAX_SIZE);
```

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 [Kiczales-Lamping92], and to apply the ideas of "reuse contracts" [Steyaert-etal96].

5 Related Work

Our general design strategy for making JML practical and effective has been to blend the Eiffel [Meyer92a] [Meyer92b] [Meyer97] and Larch [Guttag-Horning93] [LeavensLarchFAQ] [Wing87] [Wing90a] approaches to specification. From Eiffel we have used the idea that assertions are written using Java's expression syntax as much as possible, thereby avoiding large amounts of special-purpose logical notations. JML also adapts the `\old` notation from Eiffel, instead of the Larch style annotation of names with state functions. Currently JML does not come with tools to execute preconditions to help debug programs, as in Eiffel. We plan to eventually extend JML's tools to support the testing of postconditions at run-time as well.

However, Eiffel specifications, as written by Meyer, are typically not as complete as model-based specifications written, for example, in Larch BISLs or VDM [Jones90]. For example, Meyer partially specifies a `remove` (i.e., pop) method 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 (see p. 339 of [Meyer97]). However, the only characterization of which item is removed is given informally as a comment. Nothing is said formally that ensures that the other elements of the stack are unchanged. To allow more complete specifications, we need ideas from model-based specification languages.

JML's semantic differences from Eiffel (and its cousins Sather and Sather-K) allow one to more easily write more complete specifications, following the ideas of model-based specification languages. The most important of these is JML's use of specification-only declarations. These `model` declarations allow more abstract and exact specifications of behavior than is typically done in Eiffel. For example, because one has a model of the abstract values of stack objects, one can precisely state both which element is removed by `pop` and that the other elements on the stack are unchanged. The use of model fields in JML thus allows one to write specifications that are similar to the spirit of VDM or Larch BISLs.

A more minor difference from Eiffel is that in JML one can specify frame conditions, using the `assignable` clause. Our interpretation of the `assignable` clause is very strict, as even benevolent side effects are disallowed if the `assignable` clause is omitted [Leino95] [Leino95a].

Another difference from Eiffel 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++ [Leavens96b] [Leavens99] 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.

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 [Wing87], 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. (A preliminary study by Finney [Finney96] indicates that a large number of special-purpose, graphic mathematical notations, such as those found in Z [Hayes93] [Spivey92] may make such specifications hard to read, even for programmers trained in the notation.) 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 JML's semantics and for theorem proving. We believe that the use of slightly extended Java notation for assertions is appropriate, given that JML is used in detailed design, and thus will mostly be read and written by persons familiar with Java.

6 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 [Lerner91] [Sivaprasad95]. 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 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 [Wahls-Leavens-Baker98]. 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.

In conclusion, 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 as a tool for recording detailed designs.

More information about JML can be found on the web at the following URL.
'`http://www.cs.iastate.edu/~leavens/JML.html`'

Acknowledgments

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. Thanks to Anand Ganapathy for his work on the type checker used to check our specifications.

The work of Leavens and Ruby was supported in part by a grant from Rockwell International Corporation and by the US NSF under grant CCR-9503168. The work of Leavens, Baker, and Ruby is supported in part by the NSF grant CCR 9803843.

This paper is adapted from Haim Kilov, Bernhard Rumpe, and William Harvey (editors), *Behavioral Specifications for Businesses and Systems*, chapter 12, pages 175-188.

About the Authors

The authors are all at the Department of Computer Science, Iowa State University, in Ames, Iowa, 50011-1040 USA.

Gary T. Leavens is a professor. He received a Ph.D. in Computer Science from MIT in 1989. His research focuses on formal methods in OO programming, and includes the theory of abstract data types, specification, verification, as well as topics in programming languages such as type theory and semantics. He has been involved in the design of the specification language Larch/Smalltalk, and is a principal designer of both Larch/C++ and JML. He is the author of the Larch FAQ.

Albert L. Baker is an associate professor. He received a Ph.D. in Computer Science from The Ohio State University in 1979. His research focuses on specification languages, software testing, prototyping from formal specifications and CASE tools. He has been involved in the design of the specification language SPECS/C++ and is a principal designer of JML.

Clyde Ruby is a Ph.D. student. He has more than 15 years experience as an analyst, designer, and implementer of software systems. His current research focuses on formal methods in object-oriented programming, specification, and verification. He is working with Leavens on Larch/C++ and JML.

Bibliography

[Arnold-Gosling98]
Arnold, K. and Gosling, J. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.

[Borgida-Mylopoulos-Reiter95]
Borgida, A., Mylopoulos, J., and Reiter, R. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[Dhara-Leavens96]
Dhara, K. K. and Leavens, G. T. 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. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

[Finney96] Finney, K. Mathematical notation in formal specification: Too difficult for the masses? *IEEE Transactions on Software Engineering*, 22(2):158–159, February 1996.

[Guttag-Horning93]
Guttag, J. V., Horning, J. J., Garland, S., Jones, K., Modet, A., and Wing, J. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.

[Gosling-Joy-Steele96]
Gosling, J., Joy, B., and Steele, G. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, 1996.

[Hoare72a]
Hoare, C. A. R. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[Hayes93] Hayes, I., editor. *Specification Case Studies*. International Series in Computer Science. Prentice-Hall, Inc., second edition, 1993.

[Jones90] Jones, C. B. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[Kiczales-Lamping92]
Kiczales, G. and Lamping, J. Issues in the design and documentation of class libraries. *ACM SIGPLAN Notices*, 27(10):435–451, October 1992. *OOPSLA '92 Proceedings*, Andreas Paepcke (editor).

[Lerner91] Lerner, R. A. Specifying objects of concurrent systems. Ph.D. Thesis CMU-CS-91-131, School of Computer Science, Carnegie Mellon University, May 1991.

[Leino95a] Leino, K. R. M. 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.

[Leino95] Leino, K. R. M. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[Leavens96b]
Leavens, G. T. An overview of Larch/C++: Behavioral specifications for C++ modules. In Kilov, H. and Harvey, W., 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.

[LeavensLarchFAQ]
Leavens, G. T. Larch frequently asked questions. Version 1.110. Available in '`http://www.cs.iastate.edu/~leavens/larch-faq.html`', May 2000.

[Leavens99]
Leavens, G. T. Larch/C++ Reference Manual. Version 5.41. 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`', April 1999.

[Leavens-Baker99]
Leavens, G. T. and Baker, A. L. Enhancing the pre- and postcondition technique for more expressive specifications. In Wing, J. M., Woodcock, J., and Davies, J., editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Proceedings*, volume 1709 of *Lecture Notes in Computer Science*, pages 1087–1106. Springer-Verlag, 1999.

[Leavens-Baker-Ruby01]
Leavens, G. T., Baker, A. L., and Ruby, C. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06o, Iowa State University, Department of Computer Science, May 2001.

[Lano-Haughton94]
Lano, K. and Haughton, H., editors. *Object-Oriented Specification Case Studies*. The Object-Oriented Series. Prentice Hall, New York, NY, 1994.

[Liskov-Wing94]
Liskov, B. and Wing, J. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[Meyer92a]
Meyer, B. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[Meyer92b]
Meyer, B. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[Morgan94]
Morgan, C. *Programming from Specifications: Second Edition*. Prentice Hall International, Hempstead, UK, 1994.

[Meyer97] Meyer, B. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[Ogden-etal94]
Ogden, W. F., Sitaraman, M., Weide, B. W., and Zweben, S. H. Part I: The RESOLVE framework and discipline — a research synopsis. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, Oct 1994.

[Poetzsch-Heffter97]
Poetzsch-Heffter, A. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.

[Spivey92] Spivey, J. M. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, New York, NY, second edition, 1992.

[Sivaprasad95]
Sivaprasad, G. 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.

[Steyaert-etal96]
Steyaert, P., Lucas, C., Mens, K., and D'Hondt, T. Reuse contracts: Managing the evolution of reusable assets. In *OOPSLA '96 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 268–285. ACM Press, October 1996. ACM SIGPLAN Notices, Volume 31, Number 10.

[Tan94] Tan, Y. M. Interface language for supporting programming styles. *ACM SIGPLAN Notices*, 29(8):74–83, August 1994. Proceedings of the Workshop on Interface Definition Languages.

[Tan95] Tan, Y. M. *Formal Specification Techniques for Engineering Modular C Programs*, volume 1 of *Kluwer International Series in Software Engineering*. Kluwer Academic Publishers, Boston, 1995.

[Wing83] Wing, J. M. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

[Wing87] Wing, J. M. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[Wing90a] Wing, J. M. A specifier's introduction to formal methods. *Computer*, 23(9):8–24, September 1990.

[Wills94] Wills, A. Refinement in Fresco. In Lano and Houghton [Lano-Haughton94], chapter 9, pages 184–201.

[Wahls-Leavens-Baker98]
Wahls, T., Leavens, G. T., and Baker, A. L. 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.