An Overview of Larch/C++: Behavioral Specifications for C++ Modules

Gary T. Leavens

TR #96-01c
February 1996, revised March, April 1996, January 1997

Keywords: behavioral specification, model-based, behavioral interface specification language, Larch, C++, Larch/C++, Larch Shared Language, VDM, Z, correctness, verification, abstract data type, object-oriented, specification inheritance, example, checkable redundancy, behavioral subtype, informality, tunable formality.
1993 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — Languages; F.3.1 [*Logics and Meaning of Programs*] Specifying and verifying and reasoning about programs — Assertions, invariants, pre- and post-conditions, specification techniques.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

AN OVERVIEW OF LARCH/C++: BEHAVIORAL SPECIFICATIONS FOR C++ MODULES

Gary T. Leavens*
Department of Computer Science
Iowa State University, Ames, Iowa 50011 USA

January 23, 1997

Abstract

An overview is presented of the behavioral interface specification language Larch/C++. The features of Larch/C++ used to specify the behavior of C++ functions and classes, including subclasses, are described, with examples. Comparisons are made with other object-oriented specification languages. An innovation in Larch/C++ is the use of examples in function specifications.

1 Introduction

Larch/C++ [31] is a model-based specification language that allows the specification of both the exact interface and the behavior of a C++ [13, 44] program module.

1.1 Model-Based Specification

The idea of model-based specifications builds on two seminal papers by Hoare. Hoare's paper "An Axiomatic Basis for Computer Programming" [21], used two predicates over program states to specify a computation. The first predicate specifies the requirements on the state before the computation; it is called the computation's *precondition*. The second predicate specifies the desired final state; it is called the computation's *postcondition*.

Hoare's paper "Proof of correctness of data representations" [22], described the verification of abstract data type (ADT) implementations. In this paper Hoare introduced the use of an abstraction function that maps the implementation data structure (e.g., an array) to a mathematical value space (e.g., a set). The elements of this value space are thus called *abstract values* [34]. The idea is that one specifies the ADT using the abstract values, which allows clients of the ADT's operations to reason about calls without worrying about the details of the implementation.

A *model-based* specification language combines these ideas. That is, it specifies procedures (what C++ calls functions), using pre- and postconditions. The pre- and postconditions use the vocabulary specified in an *abstract model*, which specifies the abstract values mathematically.

*Leavens's work was supported in part by NSF grant CCR-9593168.

The best-known model-based specification languages are VDM-SL [23] and Z [42, 41, 19]. Both come with a mathematical toolkit from which a user can assemble abstract models for use in specifying procedures. The toolkit of VDM-SL resembles that of a (functional) programming language; it provides certain basic types (integers, booleans, characters), and structured types such as records, Cartesian products, disjoint unions, and sets. The toolkit in Z is based on set theory; it has a relatively elaborate notation for various set constructions, as well as powerful techniques for combining specifications (the schema calculus).

1.2 Larch

The work of Wing, Guttag, and Horning on Larch extends the VDM-SL and Z tradition in two directions [52, 51, 18]:

- Although a mathematical toolkit is provided [18, Appendix A], specifiers may design their own mathematical theories using the Larch Shared Language (LSL) [18, Chapter 4]. This allows users, if they desire, to create and use an abstract model at exactly the right level of abstraction; that is, one can either build an abstract model out of readily available parts, or one can build a model from scratch. Clearly, not everyone should be building models from scratch; thus it is convenient that those that do get built can be shared, even among users of different behavioral interface specification languages.
- Instead of one generic specification language, there are several behavioral interface specification languages (BISLs), each tailored to specifying modules to be written in a specific programming language. Examples include LCL [18, Chapter 5] (for C), LM3 [18, Chapter 6] (for Modula-3), Larch/Ada [17] (for Ada), Larch/CLU [52, 51] (for CLU), Larch/Smalltalk [10] (for Smalltalk) and Larch/C++.

The advantage of tailoring each BISL to a specific programming language is that one can specify both the behavior and the exact interface to be programmed [24]. This is of great practical benefit, because the details of the interface that need to be specified vary among programming languages. For example, because Larch/C++ is tailored to the specification of C++ code, it allows uses to specify the use of such C++ features as **virtual**, **const**, exception handling, and exact details of the C++ types (including distinctions between types such as `int` and `long int`, pointers and pointers to constant objects, etc.). No such details can be specified directly in a specification language such as VDM-SL or Z that is not tailored to C++. The same remark applies to object-oriented (OO) specification languages such as Z++ [27, 26], ZEST [11], Object-Z [39, 40], OOZE [1, 2, 3], MooZ [36, 37], and VDM++ [38]. However, apparently there are "variants of Fresco" [48, 50] that are "derived from C++ and Smalltalk" [49, p. 135]; these may permit more exact specification of interface details.

The remainder of this chapter gives a set of examples in Larch/C++, and then concludes with a discussion. The set of examples specifies a hierarchy of shapes that is used as a case study in the book *Object Orientation in Z* [43].

2 Quadrilaterals

To write a specification in Larch/C++, one specifies an abstract model in LSL, and then uses that to specify the C++ interface and its behavior. This section

```
Quad(Q): trait
  includes FourSidedFigure, NoContainedObjects(Q)
  Q tuple of edges: Edges, position: Vector
  implies
    QuadDesugared(Q)
```

Figure 1: The LSL trait `Quad`, which specifies an abstract model for quadrilaterals located at some particular position.

specifies the abstract model of quadrilaterals, then the abstract class `QuadShape` and the class `Quadrilateral`.

2.1 Abstract Model of Quadrilaterals

Although LSL has the power to specify abstract models "from scratch," most abstract models are built using tuples (records), sets, and other standard mathematical tools that are either built-in to LSL or found in Guttag and Horning's Handbook [18, Appendix A]. A typical example is given in Figure 1. That figure specifies a theory in LSL, using a LSL module, which is called a *trait*. This trait is named `Quad`, and has a parameter `Q`, which can be replaced by another type name when the trait is used. This trait itself includes instances of two other traits: `FourSidedFigure`, and `NoContainedObjects(Q)`. The latter of these simply says that an abstract value of type `Q` has no subobjects [31, Section 7.5]. The type `Q` itself is defined next, by using the built-in LSL **tuple of** notation. What LSL calls a tuple is a record-like value; in this case the tuple has two fields: `edges` of type `Edges` and `position` of type `Vector`. The types `Edges` and `Vector` are specified in the trait `FourSidedFigure`. Following this section of the trait `Quad`, beginning with the LSL keyword **implies**, is a statement that the theory of the trait `Quad` contains the theory of the trait `QuadDesugared(Q)`.

The **implies** section illustrates an important feature of the Larch approach: the incorporation of checkable redundancy into specifications. Such redundancy can serve as a consistency check; it can also highlight consequences of the specification for the benefit of readers, as in Figure 1. The trait `QuadDesugared`, which is shown in Figure 2, is a desugaring of Figure 1 (minus the **implies** section). This desugaring explains the LSL **tuple of** notation. In Figure 2, the signatures of the operators on `Q` values are specified in the lines following **introduces**, and their theory is specified in the lines following **asserts**. In the theory section, the **generated by** clause states that all abstract values of type `Q` are equivalent to `[`e`,`v`]` for some e and v. (This corresponds to the "no junk" principle of the initial algebra approach [15, 7, 14]. However, note that uses of this principle are specifiable in LSL; that is, although tuples are generated by default, LSL does not require that all types of abstract values be generated.) The **partitioned by** clause says that two `Q` values are equal unless they can be distinguished using the operators `__.edges` and `__.position`. (This is the opposite of what is done by default in the initial algebra approach.) Following the $\forall$ (typed as `\forall` by users) are some declarations and equations. Besides the field selectors, a tuple in LSL provides "update" operators; in this case they are `set_edges` and `set_position`. Because LSL abstract values are purely mathematical, these do not make any changes to a tuple, but produce a similar tuple with a change to the relevant field.

Another example of a trait that uses the tuple notation, but which demonstrates a bit more about LSL, is the trait `FourSidedFigure`. This trait is spec-

```
QuadDesugared(Q): trait
  includes FourSidedFigure, NoContainedObjects(Q)
  introduces
    [__,__]: Edges, Vector → Q
    __.edges: Q → Edges
    __.position: Q → Vector
    set_edges: Q, Edges → Q
    set_position: Q, Vector → Q
  asserts
    Q generated by [__,__]
    Q partitioned by __.edges, __.position
    ∀ e,e1: Edges, v,v1: Vector
      ([e,v]).edges == e;
      ([e,v]).position == v;
      set_edges([e,v], e1) == [e1,v];
      set_position([e,v], v1) == [e,v1];
```

Figure 2: The LSL trait `QuadDesguared`, which is a desugaring of the trait `Quad`.

ified in Figure 3. It includes two other traits: `PreVector` defines an abstract model of vectors (the type `Vec[T]` with an approximate `length` operator[1]) and `int` which is an abstract model of the integers (the type `int` with appropriate auxiliary definitions for C++). The trait `FourSidedFigure` defines the type `Edges` as a tuple of four `Vector` values. As a convenience, the trait also introduces the operator, `__[__]`, which allows one to write `e[1]` instead of `e.v1`. In the **asserts** section, the specification defines the condition on four-sided figures from [43] as a predicate. The predicate `isLoop(e)` holds just when the vectors sum to zero (make a loop). In the **implies** section this property is stated in an equivalent way. (It would be inconsistent (i.e., wrong) to simply assert that the edges always sum to zero; doing so would assert that all combinations of four vectors sum to zero. Such properties must be handled by either constructing an abstract model from scratch, or by asserting that the property holds at the interface level, as is done below.)

In [43], vectors are usually treated as a *given set*, meaning that their specification is of no interest. A type of values can be treated as a given set in LSL by simply specifying the signatures of its operators that are needed in other parts of the specification, without giving any assertions about their behavior. For example, to treat vectors as a given set, one would have `FourSidedFigure` include the trait `PreVectorSig`, as specified in Figure 4, instead of `PreVector`.

Although it is perfectly acceptable to treat vectors as a given set (and beginning users are encouraged to make similar simplifications to avoid mathematical difficulties), one can illustrate more of the power of LSL by fleshing out the trait `PreVector`. This is done in Figure 5. In this trait's **assumes** clause, the type `T` is required to be a ring with a unit element, have a commutative multiplication operator, be totally ordered, and to have conversions to and from the real numbers. (The first three assumed traits are found in [18, Appendix A]; the last trait, and the included trait `Real` that specifies the

[1] In the trait `FourSidedFigure`, the type `Vec[T]` is renamed to be `Vector`. The specifications in [43] are a bit vague on exactly what capabilities are needed by the scalar type (which is named `Scalar` in `FourSidedFigure` and `T` in the trait `PreVector`). As there is no easy way to implement an exact length function (because some lengths are irrational) the specification in `PreVector` allows the `length` operator to return an approximate result.

```
FourSidedFigure: trait

  includes PreVector(Scalar, Vector for Vec[T]), int

  Edges tuple of v1: Vector, v2: Vector, v3: Vector, v4: Vector

  introduces
    __[__]: Edges, int → Vector
    isLoop: Edges → Bool

  asserts ∀ e: Edges
     isLoop(e) == (e.v1 + e.v2 + e.v3 + e.v4 = 0:Vector);
     e[1] == e.v1;
     e[2] == e.v2;
     e[3] == e.v3;
     e[4] == e.v4;

  implies ∀ e: Edges
     isLoop(e) == (e[1] + e[2] + e[3] + e[4] = 0:Vector);
```

Figure 3: The LSL trait **FourSidedFigure**.

```
PreVectorSig(T): trait
  introduces
    __ + __: Vec[T], Vec[T] → Vec[T]
    __ * __: T, Vec[T] → Vec[T]
    0: → Vec[T]
    - __: Vec[T] → Vec[T]
    __ - __: Vec[T], Vec[T] → Vec[T]
    __ · __: Vec[T], Vec[T] → T      % inner product
    length: Vec[T] → T               % approximate length
```

Figure 4: The LSL trait **PreVectorSig**, which can be used if the type **Vec[T]** is to be treated as a "given".

real numbers, are found in [30].) The use of traits for stating such assumptions is similar to the way that theories are used for parameterized specifications in OBJ [16, 14]. The assertions in the trait **PreVector** specify the theory of an inner product and the approximate length function. (Comments in LSL start with **%** and continue to the end of a line.) Two features of the **implies** section not previously seen are illustrated in this trait. The naming of another trait, in this case **PreVectorSig(T)** says that the theory of that trait is included in this trait's theory. The **PreVector** trait's **converts** clause says that there is no ambiguity in the specification of the inner product operator. On the other hand, the **length** operator is not so well-specified, and thus is not named in the **converts** clause.

To push this mathematical modeling back to standard traits, one needs the trait **PreVectorSpace**, found in Figure 6. (The trait **DistributiveRingAction** is found in [30], the other traits are from [18, Appendix A].)

Now that we are done with the initial mathematical modeling, we can turn to the behavioral interface specifications.

```
PreVector(T): trait

  assumes RingWithUnit, Abelian(* for ∘),
          TotalOrder, CoerceToReal(T)

  includes PreVectorSpace(T), Real

  introduces
     __ · __: Vec[T], Vec[T] → T  % inner product
    length: Vec[T] → T

  asserts
    ∀ u,v,w: Vec[T], a, b: T

      % the inner product is bilinear
      (u + v) · w == (u · w) + (v · w);
      u · (v + w) == (u · v) + (u · w);
      (a * u) · v == a * (u · v);
      (a * u) · v == u · (a * v);

      % the inner product is symmetric (commutative)
      u · v == v · u;

      % the inner product is positive definite
      (u · u) ≥ 0;
      (u · u = 0) == (u = 0);

      approximates(length(u), sqrt(toReal(u · u)));

  implies
    PreVectorSig(T)
    converts
       __ · __: Vec[T], Vec[T] → T
```

Figure 5: The LSL trait `PreVector`.

```
PreVectorSpace(T): trait
  assumes RingWithUnit, Abelian(* for ∘)
  includes AbelianGroup(Vec[T] for T, + for ∘,
                        0 for unit, - __ for ⁻¹),
          DistributiveRingAction(T for M, Vec[T] for T)
  % ... implications omitted ...
```

Figure 6: The LSL trait `PreVectorSpace`.

2.2 Specification of QuadShape and Quadrilateral

Following the ZEST [11] and Fresco [49] specifications of the shapes example, the first class to specify is an abstract class of four-sided figures, `QuadShape`. The reason for this is that, if we follow [43, Chapter 2], then quadrilaterals are shearable, but some subtypes (rectangle, rhombus, and square) are not. If we were to follow the class hierarchy given on page 8 of [43], there would be problems, because the classes `Rectangle`, `Rhombus`, and `Square` would be subtypes but not behavioral subtypes of the types of their superclasses. Informally, a type S is a behavioral subtype of T if objects of type S can act as if they are objects of type T [4, 5, 32, 28, 35, 33]. Having subclasses not implement subtypes would make for a poor design; it would also make such classes unimplementable if specified in Larch/C++. This is because Larch/C++ forces subclasses to specify behavioral subtypes of the types of their public superclasses [12]. Thus we will follow the ZEST and Fresco specifications in using an abstract class without a `shear` operation as the superclass of `Quadrilateral`.

The Larch/C++ specification of the abstract class `QuadShape` is given in Figure 7. This behavioral interface specification includes the behavioral interface specifications of the type `Vector`, which also defines the type `Scalar`. In Larch/C++, one could also specify `QuadShape` as a C++ template class with the types `Vector` and `Scalar` as type parameters [31, Chapter 8], but the approach adopted here is more in keeping with the examples in [43].

In the specification of `QuadShape`, the first thing to note is that the syntax that is not in comments is the same as in C++. Indeed, all of the C++ declaration syntax (with a few ambiguities removed) is supported by Larch/C++. A C++ declaration form in a Larch/C++ specification means that a correct implementation must be C++ code with a matching declaration. (Hence, a Larch/C++ specification cannot be correctly implemented in Ada or Smalltalk.) This happens automatically if, as in these examples, the behavioral specifications are added as annotations to a C++ header file.

Annotations in Larch/C++ take the form of special comments. What to C++ looks like a comment of the form `//@` ... or `/*@` ... `@*/` is taken as an annotation by Larch/C++. That is, Larch/C++ simply ignores the annotation markers `//@`, `/*@`, and `@*/`; the text inside what to C++ looks like a comment is thus significant to Larch/C++.

With such annotations, the user of Larch/C++ can specify intent, semantic modeling information, and behavior. At the class level, this is done with the keywords **abstract**, **uses**, and **invariant**; at the level of C++ member function specifications this is done with the keywords **requires**, **modifies**, **trashes**, **ensures**, **example**, and **claims**.

Traits, including those that define the abstract model, are noted in **uses** clauses. In the specification of `QuadShape`, the trait used is `Quad`, with `QuadShape` replacing `Q`. In Figure 7, the **uses** clause preceeds the class defintion, so that the trait will be available to clients that include the file. (A **uses** clause within the class definition has a scope that is limited to that class.)

The use of the keyword **abstract** in the specification of the class `QuadShape`, specifies the intent that `QuadShape` is not to be used to make objects; that is, `QuadShape` is an *abstract class*. As such, it has no "constructors" and therefore no objects will exist that are direct instances of such a class. This extra information could be used in consistency checking tool [46, 45, 47].

The **invariant** clause will be explained following the explanation of the member function specifications. (As in C++, **public:** starts the public part of a class specification.)

Each member function specification looks like a C++ function declaration, followed by a specification of the function's behavior, following the keyword **behvaior**. Use of the C++ declaration syntax allows all of the C++ function declaration syntax, including **virtual** and **const**, to be used. It also allows exact C++ type information to be recorded.

To illustrate the specification format, the behavioral specification of `Move` has six clauses. The **requires** clause gives the function's precondition, the **modifies** and **trashes** clauses form a frame axiom [6], the **ensures** clause gives the function's postcondition, the **example** clause gives a redundant example of its execution, and the **claims** clause states a redundant property of the specification.

The postcondition, and the assertions in the **example** and **claims** clauses, are predicates over two states. These states are the state just before the function

```
#ifndef QuadShape_h
#define QuadShape_h

#include "Vector.h"

//@ uses Quad(QuadShape);

/*@ abstract @*/ class QuadShape {
public:
  //@ invariant isLoop(self•.edges);

  virtual Move(const Vector& v);
  //@ behvaior {
  //@   requires assigned(v, pre);
  //@   modifies self;
  //@   trashes nothing;
  //@   ensures liberally self' = set_position(self^, self^.position + v^);
  //@   example liberally ∃ e:Edges, pos: Vector
  //@                      (self^ = [e,pos] ∧ self' = [e, pos + v]);
  //@   claims liberally self'.edges = self^.edges;
  //@ }

  virtual Vector GetVec(int i) const;
  //@ behvaior {
  //@   requires between(1, i, 4);
  //@   ensures result = self^.e[i];
  //@ }

  virtual Vector GetPosition() const;
  //@ behvaior {
  //@   ensures result = self^.position;
  //@ }
};
#endif
```

Figure 7: The Larch/C++ specification of the C++ class `QuadShape`.

body's execution, called the *pre-state*, and the state just before the function body returns (or signals an exception), called the *post-state*. A C++ object (a location) can be thought of as a box, with contents that may differ in different states. The box may also be empty. When the box empty, the object is said to be *unassigned*; an object is *assigned* when it contains a proper value. C++ objects are formally modeled in Larch/C++ using various traits [31, Section 2.8], and these traits allow one to write `assigned(v, pre)` to assert that the object `v` is allocated and assigned in the pre-state. (The pre- and post-states are reified in Larch/C++ using the keywords **pre** and **post**.) There is also a more useful notation for extracting the value of an assigned object in either state. The value of an assigned object, `o`, in the pre-state is written `o^`, and the post-state value of `o` is written `o'`.

In a member function specification, the object **self** is defined to be the same as the C++ object `*this`, which is the implicit receiver of the member function call. Thus the postcondition of `Move` says that the post-state value of the receiver object is equal to the pre-state value, with the position field changed to the pre-state position plus the pre-state value of the vector `v`. (Except for constructors, the object **self** is implicitly required to be assigned in every member function of a class [31, Section 6.2.2].)

The **ensures** clause of `Move`'s specification uses the Larch/C++ keyword

liberally. This makes it a partial correctness specification; that is, the specification says that if `v` is assigned and if the execution of `Move` terminates normally, then the post-state must be such that the position field of **self**'s value is the sum of the pre-state position and `v`. However, the function need not always terminate normally; for example, it might abort the program if the numbers representing the new position would be too large.

If **liberally** is omitted, then a total correctness interpretation is used; for example, `GetPosition` must terminate normally whenever it is called. Neither VDM, Z, nor any other OO specification languages that we know of permit mixing total and partial correctness in this manner.

A function may *modify* an allocated object by changing its value from one proper value to another, or from `unassigned` to some proper value. Each object that a function is allowed to modify must be noted by that function's **modifies** clause. For example, `Move` is allowed to modify **self**. An omitted **modifies** clause means that no objects are allowed to be modified. For example, `GetVec` and `GetPosition` cannot modify any objects.

A function may *trash* an object by making it either become deallocated or by making its value be `unassigned`. The syntax **trashes nothing** means that no object can be trashed, and is the default meaning for the **trashes** clause when it is omitted, as in `GetVec` and `GetPosition`. Noting an object in the **trashes** clause allows the object be trashed, but does not mandate it (just as the **modifies** clause allows modification but does not mandate it).

Having a distinction between modification and trashing may seem counterintuitive, but is important in helping shorten the specifications users have to write. In LCL and other Larch interface languages, these notions are not separated, which this leads to semantic problems [8, 9]. By following Chalin's ideas, most Larch/C++ function specifications do not have to make assertions about objects being allocated and assigned in postconditions. This is because, if an object is modified, it must stay allocated, and if it was assigned in the pre-state, it must also be assigned in the post-state [31, Section 6.2.3].

An **example** adds checkable redundancy to a specification. There may be several examples listed in a single function specification in Larch/C++. For each example, what is checked is roughly that the example's assertion, together with the precondition should imply the postcondition [31, Section 6.7]. (The $\exists$ in the example given is typed as `\E` by users.) As far as we know, this idea of adding examples to formal function specifications is new in Larch/C++.

Another instance of the checkable redundancy idea is the **claims** clause, which is a feature of Tan's work on LCL [46, 45, 47]. This borrowing from LCL can be used to state a redundantly checkable property implied by the conjunction of the precondition and postcondition. In the example, the claim follows from the postcondition and the meaning of `set_position` (see Figures 1 and 2).

Claims, examples, and the **trashes** and **modifies** clauses, are optional in a function specification, as illustrated by the specification of `GetVec`. The specification of `GetPosition` illustrates that the **requires** clause may also be omitted; it defaults to **requires** `true`.

In the specification of `GetVec`, `i` is passed by value. Thus `i` is not considered an object within the specification. This is why `i` denotes an `int` value, and why notations such as `i^` are not used [18, Chapter 5].

The **invariant** clause describes a property that must hold in each visible state; it can be thought of as implicitly conjoined to the pre- and postconditions of each member function specification. The notation **self**$^\bullet$ stands for the abstract value of **self** in a visible state. (The glyph $^\bullet$ is typed `\any` by users.) Thus the invariant in Figure 7 says that the edges part of the abstract value of

```
#include "Scalar.h"
//@ spec class Vector;
//@ uses PreVector(Scalar, Vector for Vec[T]);
//@ uses NoContainedObjects(Vector);
```

Figure 8: This specification module illustrates how to treat the type `Vector` as a "given" type in Larch/C++.

```
#include "QuadShape.h"
#include "Shear.h"

//@ uses QuadSubtype(Quadrilateral, QuadShape);

class Quadrilateral : public QuadShape {
public:
  //@ simulates QuadShape by toSuperWithoutChange;

  Quadrilateral(Vector v1, Vector v2, Vector v3, Vector v4,
                Vector pos);
  //@ behvaior {
  //@   requires isLoop([v1,v2,v3,v4]);
  //@   modifies self;
  //@   ensures self' = [[v1, v2, v3, v4], pos];
  //@ }

  virtual void ShearBy(const Shear& s);
  //@ behvaior {
  //@   requires assigned(s, pre);
  //@   modifies self;
  //@   ensures informally "self is sheared by s";
  //@ }
};
```

Figure 9: The Larch/C++ specification of the C++ class `Quadrilateral`.

self in each visible state must form a loop.

Note that, by using model-based specifications, it is easy to specify abstract classes. One imagines that objects that satisfy the specification have abstract values, even though there are no constructors.

Finally, note that the type `Vector` does not have to be fully specified in Larch/C++ in order to be imported by the specification of `QuadShape`. It can be regarded as "given" by making a specification module for it that simply declares the type `Vector`, and uses the appropriate traits. An example of how to do this is shown in Figure 8. (The keyword **spec** says that the declaration does not have to appear exactly as stated in an implementation; an implementation might also define `Vector` with a **typedef**, or in some other way.)

The specification of the subclass `Quadrilateral` is given in Figure 9. The C++ syntax ": **public** `QuadShape`" is what says that `Quadrilateral` is a public subclass (and hence a subtype) of `QuadShape`. In Larch/C++, a subclass is forced to be a behavioral subtype of the type of its public superclass. Roughly speaking, the idea is that the specification of each virtual member function of `QuadShape` must be satisfied by a correct implementation of that virtual member function in the class `Quadrilateral`.

Technically, in Larch/C++ behavioral subtyping is forced by inheriting the

```
QuadSubtype(Sub,Super): trait
  includes Quad(Sub), Quad(Super);
  introduces
    toSuperWithoutChange: Sub → Super
  asserts ∀ x: Sub
      toSuperWithoutChange(x) == ([x.edges, x.position]):Super;
```

Figure 10: The LSL trait `QuadSubtype`.

specification of the supertype's invariant and virtual member functions in the subtype [12]. The technical problem to overcome is that the supertype's specifications were written as if **self** were a supertype object. But when applying such specifications to the subtype, **self** is a subtype object. In most OO specification languages, including Object-Z [39, 40], MooZ [36, 37], VDM++ [38], Z++ [27, 26], OOZE [1, 2, 3], and ZEST [11], there is no problem treating the subtype's abstract values as abstract values of the supertypes (and in deciding how to do that), because every object's abstract value is a tuple (i.e., a record or schema) of abstract fields; the subtype's abstract values may have more such abstract fields than the supertype's, and a subtype abstract value can be treated as a supertype value by simply ignoring the additional fields. In Larch-style BISLs, such as Larch/C++, LM3 [18, Chapter 6], and Larch/Smalltalk [10], abstract values of object do not have to be tuples. This means that there is a problem in giving a semantics to inherited specifications. In Larch/Smalltalk the problem is resolved by having the user write enough operators in a trait so that the operators used in the supertype's specification are also defined on the abstract values of the subtype. However, because that solution seems to have modularity problems [29], a slightly less general solution is currently used in Larch/C++. What Larch/C++ currently (in release 5.1) requires is that the user specify a simulation function, which maps the abstract values of subtype objects to the abstract values of supertype objects [12]. Inheritance of the supertype's specifications is accomplished by applying the simulation function to each term whose type is the supertype. For example, the specification of the member function `GetPosition` inherited by `Quadrilateral`, could be written as follows in `Quadrilateral`'s specification.

```
//@ virtual Vector GetPosition() const;
//@ behvaior {
//@   ensures result = toSuperWithoutChange(self^).position;
//@ }
```

Specifying the simulation function is the purpose of the **simulates** clause.

The trait used by `Quadrilateral` is the trait `QuadSubtype`, which is shown in Figure 10. This trait includes the trait `Quad` twice, once changing the name `Q` to the subtype, and once changing it to the supertype. It defines the simulation function `toSuperWithoutChange` that maps the subtype's values to the supertype's values.

The "constructor" specified for the class `Quadrilateral` has the same name as the class in C++. Its specification follows the **simulates** clause. Constructors in C++ really are initializers, and this constructor must set the post-state value of the object to the appropriate abstract value. The **requires** clause is needed so that the object will satisfy the invariant inherited from `QuadShape`.

The specification of `ShearBy` illustrates another feature of Larch/C++: informal predicates. An informal predicate looks like the keyword **informally**, followed by a string constant. Such a predicate can be used to suppress details about a specification. This is done frequently in the specifications in [43] by using comments instead of formal specifications when discussing shearing. This also illustrates how one can use informal predicates to "tune" the level of formality in a Larch/C++ specification. For example, in Larch/C++ one could start out by using largely informal specifications, and then increase the level of formality as needed or desired.

3 Other Subtypes of QuadShape

This section contains the behavioral interface specifications of the other subtypes of `QuadShape` described in [43].

As in [11], we start with the abstract type `ParallelShape`, which is shown in Figure 11. The **invariant** clause in this specification says that the abstract values of such objects must have edges with parallel sides. (The operator `isaParallelogram` is specified in the trait shown in Figure 12.)

An interesting aspect of `ParallelShape` (apparently overlooked in all the specifications in [43]) is that if all the sides of a quadrilateral are zero length, then the angle to be returned by `AnglePar` is not well defined. The specification of `AnglePar` illustrates how to specify exceptions to handle such cases. Note first that the body of `AnglePar` has two pairs of pre- and postcondition specifications. Larch/C++ actually permits any number of these *specification cases* in a function specification body; the semantics is that the implementation must satisfy all of them [52, Section 4.1.4] [48, 49, 50]. Thus this specification says that if the object **self** has an interior, the appropriate angle must be returned, and if not, then an exception must be thrown. Although the mathematics of angles is left informal, the specification of the exception is formalized. The term **thrown**`(NoInterior)` denotes the abstract value of the exception result

The specification of the type `NoInterior` is in Figure 13. This specification uses the Larch/C++ built-in trait `NoInformationExecption` [31, Section 6.9] to specify the abstract model of the type `NoInterior`. This trait is designed as an aid in specifying abstract models for exceptions in which no significant information is being passed; it says that there is only one abstract value: `theException`. The class specification also specifies the default constructor.

Turning to another concrete class specification, the type `Parallelogram` (see Figure 14) is a public subclass of `Quadrilateral` and `ParallelShape`. (This follows the design in [43]; whether this is a good idea for a design in C++ is debatable.) It inherits the specifications of each, including the `ShearBy` member function of `Quadrilateral`, and the invariant from `ParallelShape` (including the inherited invariant from `QuadShape`). This is done by specifying a simulation function for each supertype. Of course, the constructor of `Quadrilateral` is not inherited, and so a constructor must be specified. This specification is a partial correctness specification, which allows for cases in which the vector cannot be successfully negated.

Another shape type is `Rhombus`, which is specified in Figure 15. This class is specified as a public subclass of `ParallelShape`. The trait used to specify the operator `isaRhombus` is in Figure 16.

The class `Rectangle` is specified in Figure 17. Its invariant is specified using the trait `IsaRectangle` from Figure 18.

Finally, in Figure 19 the class `Square` is specified as a public subclass of both `Rhombus` and `Rectangle`. The trait `IsaSquare`, given in Figure 20, is used in the specification of the constructor to state a claim that follows from the inherited invariant, but which might not otherwise be obvious.

```
#ifndef ParallelShape_h
#define ParallelShape_h

#include "QuadShape.h"
#include "NoInterior.h"

//@ uses QuadSubtype(ParallelShape, QuadShape);

/*@ abstract @*/ class ParallelShape : public QuadShape {
public:
  //@ simulates QuadShape by toSuperWithoutChange;

  //@ uses IsaParallelogram;
  //@ invariant isaParallelogram(self•.edges);

  virtual double AnglePar() const throw(NoInterior);
  //@ behvaior {
  //@   requires ¬(self^.edges[1] = 0 ∨ self^.edges[2] = 0);
  //@   ensures informally "result is the angle between self^.edges[1] and"
  //@                      "self^.edges[2]";
  //@ also
  //@   requires self^.edges[1] = 0 ∨ self^.edges[2] = 0;
  //@   ensures thrown(NoInterior) = theException;
  //@ }
};
#endif
```

Figure 11: The Larch/C++ specification of the C++ class `ParallelShape`.

```
IsaParallelogram: trait
  includes FourSidedFigure
  introduces
    isaParallelogram: Edges → Bool
  asserts ∀ e: Edges
      isaParallelogram(e) == isLoop(e) ∧ (e.v1 + e.v3 = 0:Vector);
  implies ∀ e: Edges
      isaParallelogram(e) == isLoop(e) ∧ (e.v2 + e.v4 = 0:Vector);
```

Figure 12: The trait `IsaParallelogram`.

```
//@ uses NoInformationException(NoInterior),
//@      NoContainedObjects(NoInterior);

class NoInterior {
public:
  NoInterior();
  //@ behvaior {
  //@   modifies self;
  //@   ensures result = theException;
  //@ }
};
```

Figure 13: The Larch/C++ specification of the C++ class `NoInterior`.

```
#include "Quadrilateral.h"
#include "ParallelShape.h"

//@ uses QuadSubtype(Parallelogram, Quadrilateral);
//@ uses QuadSubtype(Parallelogram, ParallelShape);

class Parallelogram : public Quadrilateral, public ParallelShape {
public:
  //@ simulates Quadrilateral by toSuperWithoutChange;
  //@ simulates ParallelShape by toSuperWithoutChange;

  Parallelogram(Vector v1, Vector v2, Vector pos);
  //@ behvaior {
  //@   modifies self;
  //@   ensures liberally self' = [[v1, v2, -v1, -v2], pos];
  //@ }
};
```

Figure 14: The Larch/C++ specification of the C++ class `Parallelogram`.

```
#include "ParallelShape.h"

//@ uses QuadSubtype(Rhombus, ParallelShape);

class Rhombus : public ParallelShape {
public:
  //@ simulates ParallelShape by toSuperWithoutChange;

  //@ uses IsaRhombus;
  //@ invariant isaRhombus(self•.edges);

  Rhombus(Vector v1, Vector v2, Vector pos);
  //@ behvaior {
  //@   requires length(v1) = length(v2);
  //@   modifies self;
  //@   ensures liberally self' = [[v1, v2, -v1, -v2], pos];
  //@ }
};
```

Figure 15: The Larch/C++ specification of the C++ class `Rhombus`.

4 Discussion and Conclusions

The shapes example from [43] is perhaps not ideal for illustrating the mechanisms in Larch/C++ used for specification inheritance, as the subtypes all use isomorphic spaces of abstract values. In [12], we give more interesting examples, in which the abstract models of the subtype objects contain more information than objects of their supertypes.

However, the shapes example does permit direct comparison to the OO specification languages presented in [43]. The following are the most basic points of similarity and difference.

- The LSL traits specified in the examples correspond roughly to the Z specifications given in [43, Chapter 2]. This says that LSL is roughly comparable to Z in terms of modeling power. However, LSL includes syntax for stating redundant properties of traits, which may help catch errors in such mathematical modeling.

```
IsaRhombus: trait
  includes IsaParallelogram
  introduces
    isaRhombus: Edges → Bool
  asserts
    ∀ e: Edges
      isaRhombus(e) == isaParallelogram(e)
                       ∧ (length(e.v1) = length(e.v2));
  implies
    ∀ e: Edges
      isaRhombus(e) ⇒ isaParallelogram(e);
      isaRhombus(e) == isaParallelogram(e)
                       ∧ (length(e.v1) = length(e.v3));
      isaRhombus(e) == isaParallelogram(e)
                       ∧ (length(e.v1) = length(e.v4));
```

Figure 16: The trait **IsaRhombus**.

```
#include "ParallelShape.h"

//@ uses QuadSubtype(Rectangle, ParallelShape);

class Rectangle : public ParallelShape {
public:
  //@ simulates ParallelShape by toSuperWithoutChange;

  //@ uses IsaRectangle;
  //@ invariant isaRectangle(self•.edges);

  Rectangle(Vector v1, Vector v2, Vector pos);
  //@ behvaior {
  //@   requires v1 · v2 = 0:Vector;
  //@   modifies self;
  //@   ensures liberally self' = [[v1, v2, -v1, -v2], pos];
  //@ }
};
```

Figure 17: The Larch/C++ specification of the C++ class **Rectangle**.

```
IsaRectangle: trait
  includes IsaParallelogram
  introduces
    isaRectangle: Edges → Bool
  asserts
    ∀ e: Edges
      isaRectangle(e) == isaParallelogram(e) ∧ (e.v1 · e.v2 = 0);
  implies
    ∀ e: Edges
      isaRectangle(e) ⇒ isaParallelogram(e);
      isaRectangle(e) == isaParallelogram(e) ∧ (e.v2 · e.v3 = 0);
```

Figure 18: The trait **IsaRectangle**.

```
#include "Rhombus.h"
#include "Rectangle.h"

//@ uses QuadSubtype(Square, Rhombus);
//@ uses QuadSubtype(Square, Rectangle);

class Square : public Rhombus, public Rectangle {
public:
  //@ simulates Rhombus by toSuperWithoutChange;
  //@ simulates Rectangle by toSuperWithoutChange;

  Square(Vector v1, Vector pos);
  //@ behvaior {
  //@   uses IsaSquare;
  //@   modifies self;
  //@   ensures liberally self'.edges[1] = v1 ∧ self'.position = pos;
  //@   claims liberally isaSquare(self'.edges);
  //@ }
};
```

Figure 19: The Larch/C++ specification of the C++ class `Square`.

```
IsaSquare: trait
  includes IsaRectangle, IsaRhombus
  introduces
    isaSquare: Edges → Bool
  asserts
    ∀ e: Edges
      isaSquare(e) == isaRectangle(e) ∧ isaRhombus(e);
```

Figure 20: The trait `IsaSquare`.

- The behavioral interface specifications are roughly comparable to the various OO specifications written in the OO specification languages in [43], in particular to ZEST and Fresco. However, only for Fresco is there even a hint [49, p. 135] that it may be able to specify the C++ interface details that Larch/C++ can specify.

It is important that a formal specification language not require one to formalize every detail. By allowing one to leave some types of data, some operations, and some aspects of behavior informal, Larch/C++, like Z and other OO specification languages, allows users to focus on what is important. In LSL, informality is accomplished by omitting specifications, as in Figure 4. In Larch/C++ this can be accomplished by omitting specifications, as in Figure 8, but more fine-grained tuning is permitted by the use of the informal predicates.

Larch/C++ is a large, but expressive, specification language. Most of its size and complexity arises from the complexity of C++, which, for example, has a large and complex declaration syntax, and a large number of low-level, built-in types. Although Larch/C++ has several features that other formal specification languages do not have, these features earn their place by adding much to the expressiveness of the language. For instance, the **example** and **claims** clauses in function specifications add syntax, but they allow additional checking and also allow one to convey extra information about the meaning and intent of a specification. The **example** clause is new with Larch/C++;

the idea for the **claims** clause is due to Tan [46, 45, 47].

More important for expressiveness are some fundamental semantic ideas that, while they also add to the complexity of the language, add new dimensions to the expressiveness of the language. One semantic idea is the distinction between trashing and modification [8, 9], which places the frame axiom of Larch-style specification languages on a firm semantic foundation. In Larch/C++ one can also specify such notions as whether storage is allocated or assigned. More important, allowing the user to specify both total and partial correctness for functions gives to users a choice previously reserved by specification language designers; the use of partial correctness, for example, is necessary for succinct specification of functions that may fail due to the finiteness of various data structures [21]. Allowing the specification of several specification cases (an idea due to Wing [52, Section 4.1.4] and Wills [48, 49, 50]) is convenient for the specification of exceptions and for giving a concrete form to specification inheritance [12]. Furthermore, when combined with the ability to specify both total and partial correctness, the expressiveness of the specification language becomes much more complete [20].

When combined with the approach of behavioral interface specification, the expressive features of Larch/C++ make it a step towards the day when formal documentation of C++ class libraries will be practical and useful.

Acknowledgements

This work was supported in part by NSF grant CCR-9593168. Thanks to Adrian Fiech for suggestions and comments on an earlier draft. Thanks also to Yoonsik Cheon, who helped design Larch/C++.

References

[1] A. J. Alencar and J. A. Goguen. OOZE: An object oriented Z environment. In P. America, editor, *ECOOP '91: European Conference on Object Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 180–199. Springer-Verlag, New York, N.Y., 1991.

[2] A. J. Alencar and J. A. Goguen. OOZE. In Stepney et al. [43], pages 79–94.

[3] A. J. Alencar and J. A. Goguen. Specification in OOZE with examples. In Lano and Haughton [25], pages 158–183.

[4] 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.

[5] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, New York, N.Y., 1991.

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

[7] R. M. Burstall and J. A. Goguen. Algebras, theories and freeness: An introduction for computer scientists. In Manfred Broy and Gunther Schmidt, editors, *Theoretical Foundations of Programming Methodology: Lecture Notes of an International Summer School directed by F. L. Bauer, E. W. Dijkstra and C. A. R. Hoare*, volume 91 of *series C*, pages 329–348. D. Ridel, Dordrecht, Holland, 1982.

[8] Patrice Chalin. *On the Language Design and Semantic Foundation of LCL, a Larch/C Interface Specification Language.* PhD thesis, Concordia University, 1455 de Maisonneuve Blvd. West, Montreal, Qquebec, Canada, October 1995. Available as CU/DCS TR 95-12.

[9] Patrice Chalin, Peter Grogono, and T. Radhakrishnan. Identification of and solutions to shortcomings of LCL, a Larch/C interface specification language. In Marie-Claude Gaudel and James Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 385–404, New York, N.Y., March 1996. Springer-Verlag.

[10] Yoonsik Cheon and Gary T. Leavens. The Larch/Smalltalk interface specification language. *ACM Transactions on Software Engineering and Methodology*, 3(3):221–253, July 1994.

[11] Elspeth Cusack and G. H. B. Rafsanjani. ZEST. In Stepney et al. [43], pages 113–126.

[12] 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.

[13] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Co., Reading, Mass., 1990.

[14] Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and Jose Meseguer. Principles of OBJ2. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, January 1985.

[15] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond T. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1978.

[16] Joseph A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, September 1984.

[17] David Guaspari, Carla Marceau, and Wolfgang Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, September 1990.

[18] 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.

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

[20] Wim H. Hesselink. *Programs, Recursion, and Unbounded Choice*, volume 27 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, New York, N.Y., 1992.

[21] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

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

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

[24] Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

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

[26] K. Lano and H. Haughton. Specifying a concept-recognition system in Z++. In Lano and Haughton [25], chapter 7, pages 137–157.

[27] Kevin C. Lano. Z++. In Stepney et al. [43], pages 106–112.

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

[29] Gary T. Leavens. Inheritance of interface specifications (extended abstract). In *Proceedings of the Workshop on Interface Definition Languages*, volume 29(8) of *ACM SIGPLAN Notices*, pages 129–138, August 1994.

[30] Gary T. Leavens. LSL math traits. http://www.cs.iastate.edu/~leavens/Math-traits.html, Jan 1996.

[31] Gary T. Leavens. Larch/C++ Reference Manual. Version 5.1. 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, January 1997.

[32] 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.

[33] 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.

[34] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Press, Cambridge, Mass., 1986.

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

[36] Silvio Lemos Meira and Ana Lúcia C. Cavalcanti. MooZ case studies. In Stepney et al. [43], pages 37–58.

[37] Silvio Lemos Meira, Ana Lúcia C. Cavalcanti, and Cassio Souza Santos. The Unix filing system: A MooZ specification. In Lano and Haughton [25], chapter 4, pages 80–109.

[38] Swapan Mitra. Object-oriented specification in VDM++. In Lano and Haughton [25], chapter 6, pages 130–136.

[39] Gordon Rose. Object-Z. In Stepney et al. [43], pages 59–77.

[40] Gordon Rose and Roger Duke. An Object-Z specification of a mobile phone system. In Lano and Haughton [25], chapter 5, pages 110–129.

[41] J. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, January 1989.

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

[43] Susan Stepney, Rosalind Barden, and David Cooper, editors. *Object Orientation in Z*. Workshops in Computing. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.

[44] Bjarne Stroustrup. *The C++ Programming Language: Second Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1991.

[45] Yang Meng Tan. Formal specification techniques for promoting software modularity, enhancing documentation, and testing specifications. Technical Report 619, Massachusetts Institute of Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, Mass., June 1994.

[46] 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.

[47] 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.

[48] Alan Wills. Capsules and types in Fresco: Program validation in Smalltalk. In P. America, editor, *ECOOP '91: European Conference on Object Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 59–76. Springer-Verlag, New York, N.Y., 1991.

[49] Alan Wills. Specification in Fresco. In Stepney et al. [43], chapter 11, pages 127–135.

[50] Alan Wills. Refinement in Fresco. In Lano and Houghton [25], chapter 9, pages 184–201.

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

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

Index

abstract class, 7, 9
abstract data type (ADT), 1
abstract model, 1, 15
abstract value, 1
abstraction function, 1
Ada, 2
assigned variable, 7

behavior, 1
behavioral interface specification, 2, 15
behavioral subtype, 9

C, 2
C++, 1, 2
Chalin, 9
checkable redundancy, 3, 9
claims, 9
class, abstract, 7, 9
coercion, see simulation, 10
consistency checking, 7
constructor, 11

deallocation, 8

examples, in specification, 9
examples, in specifications, 7
exception, specification of, 11
expressiveness, 16

formality, tunable, 9
formality, tuning, 4, 11
FourSidedFigure trait, 3
frame axiom, 7
Fresco, 6

given set, 4, 9
Guttag, 2

Hoare, 1
Horning, 2

informality, 4, 9, 11, 16
inheritance, of specifications, 9
interface, 1
interface specification, behavioral, 15
invariant, 9

Larch, 2
Larch Shared Language, 2
Larch/Ada, 2
Larch/C++, 1, 2
Larch/Smalltalk, 2, 9
LCL, 2, 7, 9
LCLint, 7
liberal specification, 8
LM3, 2, 9
LSL, 2, 3

member function specification, 8
model-based, 1
model-based specification, 1
modify, 8
Modula-3, 2
MooZ, 2, 9

NoInterior trait, 12

OBJ, 4
Object-Z, 2, 9
OOZE, 2, 9

ParallelShape, 11
partial correctness, 8
post-state, 7
postcondition, 1
pre-state, 7
precondition, 1
PreVector trait, 4
PreVectorSig trait, 4
PreVectorSpace trait, 5

Quad trait, 3
Quadrilateral, 9
QuadShape, 6

redundancy, checkable, 3, 9

self, 8
simulation function, 10
Smalltalk, 2
specification case, in Larch/C++, 11
specification inheritance, 9
specifying exceptions, 11
subclass, 9
subtype, behavioral, 9
supertype, behavioral, 9

Tan, 9, 16
trait, 3
trash, 8
tunable formality, 9
tuning formality, 4

unassigned variable, 7

VDM++, 2, 9
VDM-SL, 1
Vector, 9
virtual, member function specification, 9

Wing, 2

Z, 1
Z++, 2, 9
ZEST, 2, 6, 9