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

Gary T. Leavens

TR #96-01e
February 1996, revised March, April 1996,
January, July 1997, January 1999

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 13, 1999

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++ [30] is a model-based specification language that allows the specification of both the exact interface and the behavior of a C++ [12, 46] 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" [19], 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" [20], 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* [35]. 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.

The best-known model-based specification languages are VDM-SL [21] and Z [44, 43, 17]. 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

*Leavens's work was supported in part by NSF grants CCR-9593168 and CCR-9803843.

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 [54, 53, 16]:

- Although a mathematical toolkit is provided [16, Appendix A], specifiers may design their own mathematical theories using the Larch Shared Language (LSL) [16, 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 [16, Chapter 5] (for C), LM3 [16, Chapter 6] (for Modula-3), Larch/Ada [15] (for Ada), Larch/CLU [54, 53] (for CLU), Larch/Smalltalk [9] (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 [22]. 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 users 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++ [25, 24], ZEST [10], Object-Z [41, 42], OOZE [1, 2, 3], MooZ [37, 38], and VDM++ [39]. However, apparently there are "variants of Fresco" [50, 52] that are "derived from C++ and Smalltalk" [51, p. 135]; these may permit more exact specification of interface details.

The remainder of this paper 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* [45]. An index is provided for important ideas and examples.

2 Quadrilaterals

To write a specification in Larch/C++, one specifies an abstract model and mathematical vocabulary, and then uses these to specify the behavior of a C++ interface. The C++ built-in types have models that are supplied by Larch/C++ and written directly in LSL. Users can also directly supply models for objects of their own classes in LSL, but except for classes which are intended to be "pure values" like the built-in types, it is often more convenient to specify the model of a class using several specification-only data members. These *specification variables* are used in the specification, but do not have to be implemented. They often take on values of a type which is not itself intended to be implemented. Examples of both styles of class specification will be given below.

The process of specifying an abstract model and its accompanying vocabulary is not usually completed before one begins writing the behavioral specifications. Quite often, the specifications and the mathematics evolve together, as one searches for better ways to express the desired behavior, and as one better understands what is desired.

To give some examples, this section describes the specification of the abstract class `QuadShape` and the class `Quadrilateral`. I first specified `QuadShape` with 5 specification variables: four vectors representing the sides and a vector giving the position (even this builds on previous work [45]). However, it was more convenient to think of the four edge vectors as components of some structure, so they are modeled as part of an array. Arrays are already modeled by a built-in trait of Larch/C++ [30, Section 11.7], but it was convenient to define some vocabulary (operators) for creating an array of values, and for testing to see whether the vectors in such an array make a loop. In Larch/C++, such vocabulary is specified in a LSL trait. Therefore we turn to traits and the traits used in specifying `QuadShape`. (See Section2.2 for the Larch/C++ interface specification.)

2.1 Vocabulary for Specifying 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 [16, 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 `FourSidedFigure`, and has a parameter `Scalar`, which can be replaced by another type name when the trait is used. This trait itself includes instances of: `PreVector(Scalar, Vector for Vec[T])`, `int`, and `Val_Array(Vector)`. The first of these gives part of the model for vectors, and will be discussed further below. The second of these gives a model for the C++ type `int` with appropriate auxiliary definitions for C++ [30, Section 11.1.5]. The last of these gives a model for the abstract values of C++ arrays of vectors [30, Section 11.7]. This model includes such operations as indexing into an array using the `__[__]` operator. (LSL uses the notation `__` to indicate the places where arguments to mixfix operators can be passed. For example, one can obtain the element at index 3 of an array value `e` by writing `e[3]`.) The LSL name for this type (sometimes called a *sort*) is `Arr[Vector]`.

The trait `Val_Array` has a type parameter, the type of its elements, and the actual parameter, `Vector`, replaces it. Thus the trait `Val_Array(Vector)` can be thought of as a copy of the trait `Val_Array`, with the name `Vector` replacing the formal sort parameter of `Val_Array`. The trait `PreVector(Scalar, Vector for Vec[T])` is a copy of `PreVector` with the name `Scalar` replacing its parameter (`T`), and with the name `Vector` replacing `Vec[T]`. (These replacements are done simultaneously.)

Returning to Figure 1, in the lines following `introduces`, the signatures of two operators are specified. Their theory is specified in the lines following `asserts`, and (what are intended to be) redundant consequences of this theory are specified in the lines following `implies`.

In the `asserts` section, the specification defines an operator, `isLoop`, to test whether four vectors define a four-sided figure [45, Section 2.2.3]. The term `isLoop(e)` is true just when the vectors in the array value `e` sum to zero (make a loop). It is specified using the `+` operator from the trait `PreVector(Scalar, Vector for Vec[T])`. (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 in an array value sum to zero, but array values can be constructed so that this does not hold. 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 in the interface specification of `QuadShape` below.)

In the `asserts` section, the operator `\<__,__,__,__\>` is specified to make an array value from its four arguments. It is specified using the `assign` operator on array values from the trait `Val_Array(Vector)`.

The `implies` section of Figure 1 illustrates an important feature of the Larch approach: the incorporation of checkable redundancy into specifications. The first part of the `implies` section states some theorems about the operators introduced in the `asserts` section. The

```
FourSidedFigure(Scalar): trait

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

  introduces
    isLoop: Arr[Vector] -> Bool
    \<__,__,__,__\>: Vector, Vector, Vector, Vector -> Arr[Vector]

  asserts
   \forall e: Arr[Vector], v1,v2,v3,v4:Vector
     isLoop(e) == (e[0] + e[1] + e[2] + e[3] = 0:Vector);
     \<v1,v2,v3,v4\>
        == assign(assign(assign(assign(create(4), 0,v1), 1,v2), 2,v3),
                  3,v4);

  implies
   \forall e: Arr[Vector], v1,v2,v3,v4:Vector
      size(\<v1,v2,v3,v4\>) == 4;
      (\<v1,v2,v3,v4\>)[0] == v1;
      (\<v1,v2,v3,v4\>)[1] == v2;
      (\<v1,v2,v3,v4\>)[2] == v3;
      (\<v1,v2,v3,v4\>)[3] == v4;
      allAllocated(\<v1,v2,v3,v4\>);
   converts
     isLoop:Arr[Vector] -> Bool,
     \<__,__,__,__\>: Vector, Vector, Vector, Vector -> Arr[Vector]
```

Figure 1: The LSL trait `FourSidedFigure` (file `FourSidedFigure.lsl`).

second part says that the two operators specified are well-defined relative to other operators. Such redundancy can serve as a consistency check; it can also highlight consequences of the specification for the benefit of readers. One can attempt to formally prove that the theory stated in the `implies` section follows from the rest of the specification using a theorem prover, and that may be helpful in "debugging" the specification [16, Chapter 7].

In *Object Orientation in Z* [45], 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 2, instead of `PreVector`. Comments, which start with `%` and continue to the end of a line, can be used to give some informal description of these operators if desired.

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 obtain a more precise specification (and illustrate more of the power of LSL) by fleshing out the trait `PreVector`. This is done in Figure 3.

The trait `PreVector` specifies an abstract model of vectors. This trait specifies the type `Vec[T]` with an approximate `length` operator[1]. Recall that the trait `FourSidedFigure`

[1] The specifications in *Object Orientation in Z* [45] are a bit vague on exactly what capabilities are needed by the scalar type. 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.

```
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]
    __ \cdot __: Vec[T], Vec[T] -> T
    length: Vec[T] -> T
```

Figure 2: The LSL trait `PreVectorSig`, which can be used if the type `Vec[T]` is to be treated as a "given" (file `PreVectorSig.lsl`).

copies this trait and changes the names `Vec[T]` to `Vector` and `T` to `Scalar`. Such differences in naming are common when reusing traits designed for other purposes, such as `PreVector`.

In the **assumes** clause of `PreVector`, the type `T` is required to be a ring with a unit element, have a commutative multiplication operator (`*`), be totally ordered, and have conversions to and from the real numbers. (The first three assumed traits are found in [16, Appendix A]; the last trait, and the included trait `Real` that specifies the real numbers, are found in [28].) The use of traits for stating such assumptions is similar to the way that theories are used for parameterized specifications in OBJ [14, 13]. The assertions in the trait `PreVector` specify the theory of an inner product and the approximate length function.

In the **implies** of `PreVector`, the naming of another trait, in this case `PreVectorSig(T)`, says that the theory of that trait is included in this trait's theory. `PreVector`'s **converts** clause says that there is no ambiguity in the specification of the inner product operator. However, note that 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, the trait `PreVectorSpace`, found in Figure 4, is used. (The trait `DistributiveRingAction` is found in [28], the other traits are from [16, Appendix A].)

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

2.2 Specification of QuadShape and Quadrilateral

Following the ZEST [10] and Fresco [51] 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 [45, 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 *Object Orientation in Z* [45], 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, 31, 26, 36, 32]. 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 [11]. 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 5. This behavioral interface specification includes the behavioral interface specifications of the type

```
PreVector(T): trait

  assumes RingWithUnit, Abelian(* for \circ),
          TotalOrder, CoerceToReal(T)

  includes PreVectorSpace(T), Real

  introduces
     __ \cdot __: Vec[T], Vec[T] -> T  % inner product
    length: Vec[T] -> T

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

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

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

      % the inner product is positive definite
      (u \cdot u) >= 0;
      (u \cdot u = 0) == (u = 0);

      approximates(length(u), sqrt(toReal(u \cdot u)));

  implies
    PreVectorSig(T)
    converts
       __ \cdot __: Vec[T], Vec[T] -> T
```

Figure 3: The LSL trait `PreVector` (file `PreVector.lsl`).

`Vector`, which itself includes the specification of 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 [30, Chapter 8], but the approach adopted here is more in keeping with the examples in *Object Orientation in Z* [45].

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++.

```
PreVectorSpace(T): trait

  assumes RingWithUnit, Abelian(* for \circ)

  includes AbelianGroup(Vec[T] for T, + for \circ,
                        0 for unit, - __ for \inv),
           DistributiveRingAction(T for M, Vec[T] for T)

  implies

    AC(+ for \circ, Vec[T] for T), Idempotent(- __, Vec[T])

    \forall u,v,w: Vec[T], a, b: T

      % the usual axioms, apart from the abelian group axioms
      a * (u + v) == (a * u) + (a * v);
      (a + b) * u == (a * u) + (b * u);
      (a * b) * u == a * (b * u);
      1 * u == u;
      u - v == u + (- v);

      % some standard theorems
      (u + v = u + w) => v = w;
      0 * u == 0:Vec[T];
      -(a * u) == (-a) * u;
      -(a * u) == a * (-u);
      (-a) * (-u) == a * u;
      (a \neq 0 /\ a * u = a * v) => u = v;

    converts
      0: -> Vec[T],
      __+__: Vec[T], Vec[T] -> Vec[T],
      __*__: T, Vec[T] -> Vec[T],
      - __: Vec[T] -> Vec[T],
      __ - __: Vec[T], Vec[T] -> Vec[T]
```

Figure 4: The LSL trait `PreVectorSpace` (file `PreVectorSpace.lsl`).

With such annotations, the user of Larch/C++ can specify 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 `behavior`, `requires`, `requires redundantly`, `modifies`, `trashes`, `ensures`, `example`, and `ensures redundantly`.

Traits that define the vocabulary used in the specification are noted in `uses` clauses. In the specification of `QuadShape`, the trait used is `FourSidedFigure`, In Figure 5, the `uses` clause precedes the class definition, so that the trait will be available to clients that include `QuadShape.h`. (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

```
#ifndef QuadShape_h
#define QuadShape_h

#include "Vector.h"

//@ uses FourSidedFigure;

/*@ abstract @*/ class QuadShape {
public:
  //@ spec Vector edges[4];
  //@ spec Vector position;
  //@ invariant isLoop(edges\any);

  virtual Move(const Vector& v) throw();
  //@ behavior {
  //@   requires assigned(v, pre);
  //@   requires redundantly assigned(edges, pre)
  //@              /\ assigned(position, pre) /\ isLoop(edges^);
  //@   modifies position;
  //@   trashes nothing;
  //@   ensures liberally position' = position^ + v^;
  //@   ensures redundantly liberally edges' = edges^;
  //@   example liberally position^ = 0:Vector /\ position' = v^;
  //@ }

  virtual Vector GetVec(int i) const throw();
  //@ behavior {
  //@   requires between(1, i, 4);
  //@   ensures result = edges^[i-1];
  //@   example i = 1 /\ result = edges^[0];
  //@ }

  virtual Vector GetPosition() const throw();
  //@ behavior {
  //@   ensures result = position^;
  //@ }
};
#endif
```

Figure 5: The Larch/C++ specification of the C++ class `QuadShape` (file `QuadShape.h`).

are direct instances of such a class. This extra information could be used in consistency checking tools [48, 47, 49].

The specification variables `edges` and `position` together describe the abstract model of `QuadShape` values. (As in C++, `public:` starts the public part of a class specification.) As noted above, because they use the keyword `spec`, they do not need to be implemented. (An alternative specification would be to use a LSL trait that described the abstract model directly. This was done in the original version of this paper [29], and can be seen in other examples below.)

The `invariant` clause will be explained following the explanation of the member function specifications.

Each member function specification looks like a C++ member function declaration, followed by a specification of the function's behavior. The the keyword `behavior` starts the behavioral part. As previously mentioned, use of the C++ declaration syntax allows all of the C++ function declaration syntax, including `virtual`, `const`, and `throw` to be used. It also allows exact C++ type information to be recorded.

To illustrate most of the specification format, the behavioral specification of `Move` has seven clauses. The `requires` clause gives the function's precondition, the `requires redundantly` clause states a redundant property that must hold when the function is called, 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 `ensures redundantly` clause states a redundant property of the specification.

The postcondition, and the assertions in the `example` and `ensures redundantly` clauses, are predicates over two states. These states are the state just before the function body's execution, called the *pre-state*, and the state just before the function body returns (or throws 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 [30, Section 2.8], and these traits allow one to write `assigned(v, pre)`, as in the precondition of `Move`, 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^` (or `o\pre`), and the post-state value of `o` is written `o'` (or `o\post`).

Within a member function specification, data members of class instances are objects, as are arguments passed by reference. This includes specification variables. For example, in `QuadShape`, both `edges` and `position` are considered to be objects. Thus the postcondition of `Move` says that the post-state value of `position` is equal to the pre-state value of `position` plus the pre-state value of the vector `v`.

The `requires redundantly` clause is an example of checkable redundancy [48, 47, 49]. It allows one to state preconditions that will be true because they are implied by the stated precondition, the invariant, or by the semantics of Larch/C++. What would be checked is that the given precondition, and the relevant theory of the invariants and parts of the semantics of Larch/C++ imply the stated assertion. For example, the first two conjuncts of `Move`'s `requires redundantly` clause say that the specification variables `edges` and `position` are assigned in the pre-state. (The notation `/\` means "and".) This highlights the semantics of Larch/C++, which implicitly requires that all data members be assigned in visible states [30, Section 6.2.2]. (The pre-state of a constructor, and the post-state of a destructor are not considered visible states, and so are exempt from this requirement.) The last conjunct says that the invariant holds in the pre-state. The `requires redundantly` clause is new with Larch/C++.

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, then the post-state must be as specified. However, the function need not always terminate; for example, it might abort the program if the numbers representing the new position would be too large to represent.

If `liberally` is omitted, then a total correctness interpretation is used; for example, `GetPosition` must terminate whenever it is called. (Throwing an exception is considered termination, only abortion of the program or infinite loops constitute nontermination.) Neither VDM, Z, nor any other OO specification language 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[2]. For example, `Move` is allowed to modify `position`. 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 older versions of LCL and other Larch interface languages, these notions were not separated, which led to semantic problems [7, 8]. By following Chalin's ideas, most Larch/C++ function specifications do not have to make assertions about whether objects are allocated and assigned in postconditions. This is because, unless an object is named in the `trashes` clause, it must remain allocated if it was allocated in the pre-state, and if it was assigned in the pre-state, then it must also remain assigned in the post-state [30, Section 6.2.3].

An `example` clause 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 [30, Section 6.8]. 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 `ensures redundantly` clause. This is taken from Tan's work on LCL [48, 47, 49], where the idea of this kind of checkable redundancy in behavioral interface specifications first appeared. A `ensures redundantly` clause can be used to state a redundantly checkable property implied by the conjunction of the precondition, the contributions to the postcondition of the frame axioms, and the postcondition. In this case, the claim follows from the frame axioms, as `edges` cannot be modified.

All of these parts of a function's behavioral specification are optional, except for the `ensures` clause. This is illustrated by the specification of `GetPosition`. When the `requires` clause is omitted it defaults to `requires true`. (Of course, one can also omit the behavioral specification as a whole.)

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 [16, Chapter 5].

The `invariant` clause (found just before `Move`'s specification) describes a property that must be true of each assigned object of type `QuadShape` in each visible state [40]; it can also be thought of as restricting the space of values for the class. The notation `edges\any` stands for the abstract value of `edges` in any visible state. Thus the invariant in Figure 5 says that the value of the `edges` array 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. The use of specification variables (or LSL traits) allows one to describe the abstract values, even though an implementation might have no data members. One can think of these as describing the abstract values of concrete subtype objects.

The type `Vector` does not have to be fully specified in Larch/C++ in order to be included in 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 6. (Since the class `Vector` is declared using the keyword `spec`, an implementation does not have to declare it as a class,

[2]Following Leino's work [34], if an object o mentioned in a `modifies` clause has been declared to depend on some other object o', then o' may also be modified. The same qualification is also applied to the `trashes` clause. But this qualification is not needed to understand the examples in this paper.

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

Figure 6: This shows how to treat `Vector` as a "given" type in Larch/C++ (file `Vector.h`).

```
//@ spec class Scalar;
//@ uses Scalar;
```

Figure 7: Treating `Scalar` as a "given" type in Larch/C++ (file `Scalar.h`).

```
Scalar: trait
  assumes RingWithUnit(Scalar for T), Abelian(* for \circ, Scalar for T),
          TotalOrder(Scalar for T)
  includes CoerceToReal(Scalar)
```

Figure 8: The LSL trait `Scalar` (file `Scalar.lsl`).

but might define `Vector` with a `typedef` or in some other way.) Using a trait, in this case `PreVector(Scalar, Vector for Vec[T])` that specifies something with the same name (`Vector`), tells Larch/C++ about the abstract model of `Vector`. That is, if `v` is a `Vector` object, then the abstract value of `v` in some state is described by the type `Vector` specified in the used traits.

In Larch/C++, several syntactic sugars depend on the ability to extract objects that may be within an abstract value. The operator `contained_objects` must be specified for each type whose abstract values are explicitly modeled by the user in order to make these sugars work. For types like `Vector`, whose abstract values do not contain any objects, one can do this by simply including an instance of the trait `NoContainedObjects` [30, Section 7.5] to specify `contained_objects` in a way that says there are no subobjects in the abstract values.

The same trick for treating `Vector` as a given type is also used for the type `Scalar`. Its specification is given in Figure 7. The trait that it uses is given in Figure 8.

The specification of the subclass `Quadrilateral` is given in Figure 9. The C++ syntax "`: virtual public QuadShape`" is what says that `Quadrilateral` is a public subclass (and hence a subtype) of `QuadShape`. (The keyword `virtual`, in C++, allows multiple inheritors to share a single copy of the data members; this also holds in Larch/C++ for specification variables.) 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 specification of the supertype's invariant and virtual member functions in the subtype [11]. Since we have used specification variables in this example, and since these are inherited as in C++, the virtual member function specifications of the supertype, `QuadShape`, are easy to apply to the subtype. In such examples, one just forgets about extra data members in the subtype when interpreting part of a specification inherited from the supertype. This is also the case

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

class Quadrilateral : virtual public QuadShape {
public:
  Quadrilateral(Vector v1, Vector v2, Vector v3, Vector v4,
                Vector pos) throw();
  //@ behavior {
  //@   requires isLoop(\<v1,v2,v3,v4\>);
  //@   modifies edges, position;
  //@   ensures liberally edges' = \<v1,v2,v3,v4\> /\ position' = pos;
  //@ }

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

Figure 9: The Larch/C++ specification of the C++ class `Quadrilateral` (file `Quadrilateral.h`).

in other OO specification languages, including Object-Z [41, 42], MooZ [37, 38], VDM++ [39], Z++ [25, 24], OOZE [1, 2, 3], and ZEST [10].

(In Larch/C++, and in other Larch-style BISLs, such as LM3 [16, Chapter 6], and Larch/Smalltalk [9], abstract models do not have to be given by specification variables[3]. For example, in Larch/C++, one can specify a supertype and a subtype and give both of them arbitrary models by writing an LSL trait for each. In such a case, when one does not use specification variables to describe the abstract models of both the subtype and the supertype, giving a semantics to inherited specifications is a problem. See our other work [9, 27, 11] for how to handle such uncommon cases, and the Larch/C++ reference manual [30] for the details in Larch/C++.)

The "constructor" specified for the class `Quadrilateral` has the same name as the class in C++. Constructors in C++ really are initializers, and this constructor must set the post-state values of the specification variables 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 terms. An *informal term* starts with the keyword `informally` and is followed by one or more string constants. An informal term can be used anywhere a boolean term can be used. An informal term can thus be used to selectively suppress details about a specification. This suppression of detail is done frequently in the specifications in *Object Orientation in Z* [45] by using comments instead of formal specifications when discussing shearing. The use of informal terms is similar, but more tightly integrated, since informal terms can appear within more complex predicates. This example 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. Informal terms, and their tight integration into Larch/C++,

[3] Indeed, Larch/C++ is the only Larch-style BISL for which abstract models *can* be given by using specification variables.

```
//@ spec class Shear;
//@ uses NoContainedObjects(Shear);  //see a book on computer graphics
```

Figure 10: The specification of `Shear` (file `Shear.h`).

is something that is new with Larch/C++ [29].

The type `Shear` is specified as a given set in Figure 10. In this example, no signature is given for the trait functions that operate on the type `Shear`, because that type is only used informally. Using the trait `NoContainedObjects(Shear)` is enough to tell Larch/C++ that the abstract values are explicitly specified.

3 Other Subtypes of QuadShape

This section contains the specifications of the other subtypes of `QuadShape` described in *Object Orientation in Z* [45].

As in the ZEST specification of the shapes examples [10], 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 *Object Orientation in Z* [45]) 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. (By specifying an exception, the normal case is allowed to have a stronger precondition, and hence its precondition can protect the postcondition from undefinedness [33].) 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 [54, Section 4.1.4] [50, 51, 52] and that a caller must establish the disjunction of the preconditions. Thus this specification says that if the receiver's edges describe a shape with an interior, the appropriate angle must be returned, and if not, then it must throw the exception `NoInterior`. (The notations `~` and `\/` mean "not" and "or" respectively.) Although the mathematics of angles is left informal, the specification of the exception is formalized. The term `returns` is true just when the function returns normally without throwing any exception, and `throws(NoInterior)` is true just when the function throws an exception of type `NoInterior`. The claim in the exceptional case shows how one can describe the abstract value of the exception result of a given type. (In this case the claim is trivially true, because there are no other proper values and the exception result is passed by value.)

The specification of the type `NoInterior` is in Figure 13. This specification uses an instance of the Larch/C++ built-in trait `NoInformationExecption` [30, Section 6.10] to specify the abstract model of the type `NoInterior`. This trait is designed as an aid in specifying abstract models for exception types 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. In Larch/C++, the keyword `self` denotes the object that C++ programs refer to as `*this`; that is, `self` is a name for the object being constructed (or the object receiving a message in a member function). In a specification where the abstract model is given explicitly by an LSL trait (in this case by the first trait used), the value of `self'` is one of the values described in that trait.

Turning to another concrete class specification, the type `Parallelogram` (see Figure 14) is a public subclass of both `Quadrilateral` and `ParallelShape`. (This follows the design

```
#ifndef ParallelShape_h
#define ParallelShape_h

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

/*@ abstract @*/ class ParallelShape : virtual public QuadShape {
public:
  //@ uses IsaParallelogram(Scalar);
  //@ invariant isaParallelogram(edges\any);

  virtual double AnglePar() const throw(NoInterior);
  //@ behavior {
  //@   requires ~(edges^[0] = 0 \/ edges^[1] = 0);
  //@   ensures returns
  //@        /\ informally "result is the angle between edges^[0] and"
  //@                      "edges^[1]";
  //@ also
  //@   requires edges^[0] = 0 \/ edges^[1] = 0;
  //@   ensures throws(NoInterior);
  //@   ensures redundantly thrown(NoInterior) = theException;
  //@ }
};
#endif
```

Figure 11: The Larch/C++ specification of the C++ class `ParallelShape` (file `ParallelShape.h`).

```
IsaParallelogram(Scalar): trait
  includes FourSidedFigure(Scalar)
  introduces
    isaParallelogram: Arr[Vector] -> Bool
  asserts \forall e: Arr[Vector]
      isaParallelogram(e) == isLoop(e) /\ (e[0] + e[2] = 0:Vector);
  implies \forall e: Arr[Vector]
      isaParallelogram(e) == isLoop(e) /\ (e[1] + e[3] = 0:Vector);
```

Figure 12: The LSL trait `IsaParallelogram` (file `IsaParallelogram.lsl`).

in *Object Orientation in Z* [45]; 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.

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

class NoInterior {
public:
  NoInterior() throw();
  //@ behavior {
  //@   modifies self;
  //@   ensures self' = theException;
  //@ }
};
```

Figure 13: The Larch/C++ specification of the C++ class `NoInterior` (file `NoInterior.h`).

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

class Parallelogram : public Quadrilateral, public ParallelShape {
public:

  Parallelogram(Vector v1, Vector v2, Vector pos) throw();
  //@ behavior {
  //@   modifies edges, position;
  //@   ensures liberally edges' = \<v1,v2,-v1,-v2\>
  //@                      /\ position' = pos;
  //@ }
};
```

Figure 14: The Larch/C++ specification of the C++ class `Parallelogram` (file `Parallelogram.h`).

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.

4 Discussion and Conclusions

The shapes example from *Object Orientation in Z* [45] is perhaps not ideal for illustrating the mechanisms in Larch/C++ used for specification inheritance, as the subtypes all use the same set of specification variables and no member function specifications are strengthened. In our other work [11, 30], 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 *Object Orientation in Z* [45]. The following are the most basic

```
#include "ParallelShape.h"

class Rhombus : virtual public ParallelShape {
public:
  //@ uses IsaRhombus;
  //@ invariant isaRhombus(edges\any);

  Rhombus(Vector v1, Vector v2, Vector pos) throw();
  //@ behavior {
  //@   requires length(v1) = length(v2);
  //@   modifies edges, position;
  //@   ensures liberally edges' = \<v1,v2,-v1,-v2\>
  //@                     /\ position' = pos;
  //@ }
};
```

Figure 15: The Larch/C++ specification of the C++ class `Rhombus` (file `Rhombus.h`).

```
IsaRhombus: trait
  includes IsaParallelogram
  introduces
    isaRhombus: Arr[Vector] -> Bool
  asserts
    \forall e: Arr[Vector]
      isaRhombus(e) == isaParallelogram(e)
                       /\ (length(e[0]) = length(e[1]));
  implies
    \forall e: Arr[Vector]
      isaRhombus(e) => isaParallelogram(e);
      isaRhombus(e) == isaParallelogram(e)
                       /\ (length(e[0]) = length(e[2]));
      isaRhombus(e) == isaParallelogram(e)
                       /\ (length(e[0]) = length(e[3]));
```

Figure 16: The LSL trait `IsaRhombus` (file `IsaRhombus.lsl`).

points of similarity and difference.

- The LSL traits specified in the examples correspond roughly to the Z specifications given in Chapter 2 of *Object Orientation in Z* [45]. 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.

- The behavioral interface specifications are roughly comparable to the various OO specifications written in the OO specification languages in *Object Orientation in Z* [45], in particular to ZEST and Fresco. However, only for Fresco is there even a hint [51, p. 135] that it may be able to specify the C++ interface details that Larch/C++ can specify.

```
#include "ParallelShape.h"

class Rectangle : virtual public ParallelShape {
public:
  //@ uses IsaRectangle;
  //@ invariant isaRectangle(edges\any);

  Rectangle(Vector v1, Vector v2, Vector pos) throw();
  //@ behavior {
  //@   requires v1 \cdot v2 = 0:Vector;
  //@   modifies edges, position;
  //@   ensures liberally edges' = \<v1,v2,-v1,-v2\>
  //@                   /\ position' = pos;
  //@ }
};
```

Figure 17: The Larch/C++ specification of the C++ class `Rectangle` (file `Rectangle.h`).

```
IsaRectangle: trait
  includes IsaParallelogram
  introduces
    isaRectangle: Arr[Vector] -> Bool
  asserts
    \forall e: Arr[Vector]
      isaRectangle(e) == isaParallelogram(e) /\ (e[0] \cdot e[1] = 0);
  implies
    \forall e: Arr[Vector]
      isaRectangle(e) => isaParallelogram(e);
      isaRectangle(e) == isaParallelogram(e) /\ (e[1] \cdot e[2] = 0);
```

Figure 18: The trait `IsaRectangle` (file `IsaRectangle.lsl`).

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 2. In Larch/C++ informality can also be accomplished by omitting specifications as in Figure 6, 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 **`requires redundantly`**, **`example`**, and **`ensures redundantly`** clauses in function specifications add syntax, but they allow additional checking and also allow one to convey extra

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

class Square : public Rhombus, public Rectangle {
public:

  Square(Vector v1, Vector pos) throw();
  //@ behavior {
  //@   uses IsaSquare;
  //@   modifies edges, position;
  //@   ensures liberally edges'[1] = v1 /\ position' = pos;
  //@   ensures redundantly liberally isaSquare(edges');
  //@ }
};
```

Figure 19: The Larch/C++ specification of the C++ class `Square` (file `Square.h`).

```
IsaSquare: trait
  includes IsaRectangle, IsaRhombus
  introduces
    isaSquare: Arr[Vector] -> Bool
  asserts
    \forall e: Arr[Vector]
      isaSquare(e) == isaRectangle(e) /\ isaRhombus(e);
```

Figure 20: The LSL trait `IsaSquare` (file `IsaSquare.lsl`).

information about the meaning and intent of a specification. The **`requires redundantly`** and **`example`** clauses are new with Larch/C++; the idea for the **`ensures redundantly`** clause and redundancy in interface specifications is due to Tan [48, 47, 49]. (Tan called this a **`claims`** clause.)

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 [7, 8], 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 [19]. Allowing the specification of several specification cases (an idea due to Wing [54, Section 4.1.4] and Wills [50, 51, 52]) is convenient for the specification of exceptions and for giving a concrete form to specification inheritance [11]. Furthermore, when combined with the ability to specify both total and partial correctness, the expressiveness of the specification language becomes much more complete [18].

The Larch approach of behavioral interface specification [54, 53], and the expressive features of Larch/C++ make it a step towards the more practical and useful formal documentation for object-oriented classes.

Acknowledgements

This work was supported in part by NSF grants CCR-9503168 and CCR-9803843. Thanks to Adrian Fiech and Rustan Leino for suggestions and comments on earlier drafts. Rustan's comments led to several improvements in the semantics. Thanks also to Yoonsik Cheon, who helped design Larch/C++, and to Matt Markland who helped implement the Larch/C++ checker.

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. [45], pages 79–94.

[3] A. J. Alencar and J. A. Goguen. Specification in OOZE with examples. In Lano and Haughton [23], 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] Alex Borgida, John Mylopoulos, and Rayomnd Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[7] 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, Quebec, Canada, October 1995. Available as CU/DCS TR 95-12, from the URL ftp://ftp.cs.concordia.ca/pub/chalin/tr.ps.Z.

[8] 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.

[9] 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.

[10] Elspeth Cusack and G. H. B. Rafsanjani. ZEST. In Stepney et al. [45], pages 113–126.

[11] 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. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

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

[13] 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.

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

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

[16] 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.

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

[18] 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.

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

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

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

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

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

[24] K. Lano and H. Haughton. Specifying a concept-recognition system in Z++. In Lano and Haughton [23], chapter 7, pages 137–157.

[25] Kevin C. Lano. Z++. In Stepney et al. [45], pages 106–112.

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

[27] 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.

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

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

[30] Gary T. Leavens. Larch/C++ Reference Manual. Version 5.39. 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.

[31] 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.

[32] 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.

[33] Gary T. Leavens and Jeannette M. Wing. Protective interface specifications. Technical Report 96-04d, Iowa State University, Department of Computer Science, September 1997. In Michel Bidoit and Max Dauchet (editors), TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France. Volume 1214 of Lecture Notes in Computer Science, Springer-Verlag, 1997, pages 520-534. Available by anonymous ftp from ftp.cs.iastate.edu or by e-mail from almanac@cs.iastate.edu.

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

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

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

[37] Silvio Lemos Meira and Ana Lúcia C. Cavalcanti. MooZ case studies. In Stepney et al. [45], pages 37–58.

[38] Silvio Lemos Meira, Ana Lúcia C. Cavalcanti, and Cassio Souza Santos. The Unix filing system: A MooZ specification. In Lano and Haughton [23], chapter 4, pages 80–109.

[39] Swapan Mitra. Object-oriented specification in VDM++. In Lano and Haughton [23], chapter 6, pages 130–136.

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

[41] Gordon Rose. Object-Z. In Stepney et al. [45], pages 59–77.

[42] Gordon Rose and Roger Duke. An Object-Z specification of a mobile phone system. In Lano and Haughton [23], chapter 5, pages 110–129.

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

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

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

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] Alan Wills. Specification in Fresco. In Stepney et al. [45], chapter 11, pages 127–135.

[52] Alan Wills. Refinement in Fresco. In Lano and Houghton [23], chapter 9, pages 184–201.

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

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

Index