The Larch / Smalltalk Interface Specification Language

YOONSIK CHEON and GARY T. LEAVENS
Iowa State University

Object-oriented programming languages, such as Smalltalk, help to build reusable program modules. The reuse of program modules requires adequate documentation—formal or informal. Larch/Smalltalk is a formal specification language for specifying such reusable Smalltalk modules. Larch/Smalltalk firmly separates specification from implementation. In Larch/Smalltalk the unit of specification is an abstract data type, which is an abstraction of the behavior produced by one or more Smalltalk classes. A type can be a subtype of other types, which allows types to be organized based on specified behavior, and also allows for inheritance of their specifications. Larch/Smalltalk specifications are developed using specification tools integrated in the Smalltalk programming environment.

Categories and Subject Descriptors: 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 postconditions*; *specification techniques*

General Terms: Design, Documentation, Languages

Additional Key Words and Phrases: Formal methods, interface specification, Larch/Smalltalk, Smalltalk, specification inheritance, subtype, verification

1. INTRODUCTION

Object-oriented techniques encourage code reuse and modular design. In Smalltalk [Goldberg and Robson 1983], code reuse is achieved by defining one class to be a *subclass* of another class, called its *superclass*; the *subclass* inherits its data definitions and methods, or extends an existing class by adding new data definitions or new methods. To facilitate code reuse, the Smalltalk system provides a huge number of reusable library classes. The library is not fixed; it is constantly evolving as users write new classes and methods or acquire them from others. Using this library, users can develop applications with high productivity. To reuse the extended existing classes, however, users need to understand their interfaces and behavior precisely. Unfortunately, this is a hard task. One reason is that the original intention of

The work of both authors is supported in part by the National Science Foundation under the Grant CCR-9108654.
Authors' addresses: Department of Computer Science, 226 Atanasoff Hall, Iowa State University, Ames, Iowa 50011-1040; email: cheon@cs.iastate.edu and leavens@cs.iastate.edu.

implementors is not formally described anywhere. To infer it, one must read the code, but in the code it is often difficult to distinguish essential from accidental aspects. Smalltalk programs, moreover, are particularly hard to understand by just reading the code. Some reasons are the following:

—Since type checking is dynamic, it is hard to tell what kind of object a method expects as its arguments and what kind of object it returns as its result. The use of message passing, a kind of dynamic overloading, makes type interface difficult—for both computer and human readers.
—An abstraction is often spread across several classes for the sake of code reuse. For example, Booleans are implemented by three classes: Boolean, True, and False, where both True and False are subclasses of Boolean.
—Subclass relationships are usually structured to give a high degree of code sharing, rather than according to conceptual relationships.
—There are simply too many classes and methods that interact with one another. In many cases, (abstract) superclasses depend on yet-to-be-known subclasses methods, which sometimes require the user to read subclasses to understand superclasses. The ParcPlace Objectworks \ Smalltalk system contains in excess of 280 classes with over 2,000 methods. This makes it difficult to keep all the necessary details in mind when reading the code.

These considerations argue for stating both the interfaces and the behavioral characteristics of Smalltalk code in abstract terms, so that one may understand and reuse existing modules without inspecting the code itself. The user of an object-oriented system wants to understand the relationships between classes and the operations relevant to an instance without having to study their implementation. The description must be abstract so that irrelevant implementation choices and details are not exposed to clients. In short, in an environment supporting reusability, we need the abstraction that can be obtained by specification. Using a formal specification language increases precision and avoids unintended ambiguity.

In Larch/Smalltalk [Cheon 1991], we have combined Larch-style specifications [Guttag and Horning 1993] and the notion of subtyping. The unit of specification is called an *abstract data type* (or *type*, for short). A type is an abstraction of one or more Smalltalk classes. A type specification consists of a set of method specifications. The *interface* (its arguments and their types) and behavior of each method are precisely specified. The behavior of a method is specified by Hoare-style pre- and postconditions [Hoare 1969]. The vocabulary for specifying pre- and postconditions comes from the *used trait*, specified in a mostly equational style in the Larch Shared Language (LSL) [Guttag and Horning 1993]. The used trait describes the underlying mathematical model for the specified type. Having such a mathematical model allows one to reason about Smalltalk code without delving into the details of an object's implementation [Hoare 1972] (e.g., one does not need to know what its instance variables are). The mathematical model gives each object an *abstract value* in a given program state. To model mutation (e.g., assignment to

instance variables), the object's abstract value may change from one state to another.

A type can be specified to be a *subtype* of some other types, called its *supertypes*. We distinguish subtyping from subclassing in that a subtype relationship is a behavioral relationship, based on type specifications, while a subclass relationship is a code relationship. Subtyping is like inheritance of behavior, while subclassing is inheritance of code. In Larch/Smalltalk a type can be a subtype of more than one supertype, while in Smalltalk each class has only one superclass. To allow sound reasoning about programs that use subtypes, each object of such a subtype should behave like some object of each of its supertypes [Leavens 1991; America 1991]. However, as verifying such behavioral constraints is more properly part of a verification logic than of a specification language, Larch/Smalltalk does not require specified subtype relationships to be proven to be behaviorally correct. Therefore, in practice, the subtype relationships stated in Larch/Smalltalk are used for organizing specifications and for inheritance of specifications. Organizing specifications according to subtype relationships allows us to see types based on their conceptual relationships. This makes it easier for specifiers to maintain large volumes of specifications and for clients to navigate through specifications for possible reuse of program modules [LaLonde et al. 1986; LaLonde 1989].

A Larch/Smalltalk type can be parameterized by *type parameters* to specify a set of related types. Type parameters can be restricted to subtypes of given types [Cardelli and Wegner 1985].

The process of writing formal specifications is as error prone as the process of programming. As programming tools are of great help to programmers, specification tools, such as syntax and type checkers, will be a great help to specifiers. They help specifiers to check and maintain the consistency of formal text and to manage large numbers of specifications. Larch/Smalltalk provides specification browsers integrated in the Smalltalk programming system with a functionality similar to the Smalltalk class browsers. Figure 1 shows Larch/Smalltalk specification browsers [Cheon and Leavens 1994]. The main browsers, called *type browsers*, allow us to view, enter, modify, delete, and check Larch/Smalltalk specifications, and *trait browsers* allow us to browse traits written in LSL, either directly from type browsers or independently. Implementations (Smalltalk classes and methods) of currently viewed specifications can be browsed by making proper selections on type browsers. Like Smalltalk code, Larch/Smalltalk specifications are not just plain text, but organized material accessed through specification browsers. These tools allow specifications to be developed and used practically in the programming process.

Our experience shows that unambiguous, precise, and abstract descriptions of Smalltalk modules in Larch/Smalltalk can be written. Such specifications would be the necessary starting point for formal verification, although formal verification of Smalltalk is outside the scope of this paper.

In the next section, we give a short introduction to Larch-style specifications with a brief overview of LSL. In Section 3 we introduce language constructs for specifying simple types. In Section 4, we formalize the notion of

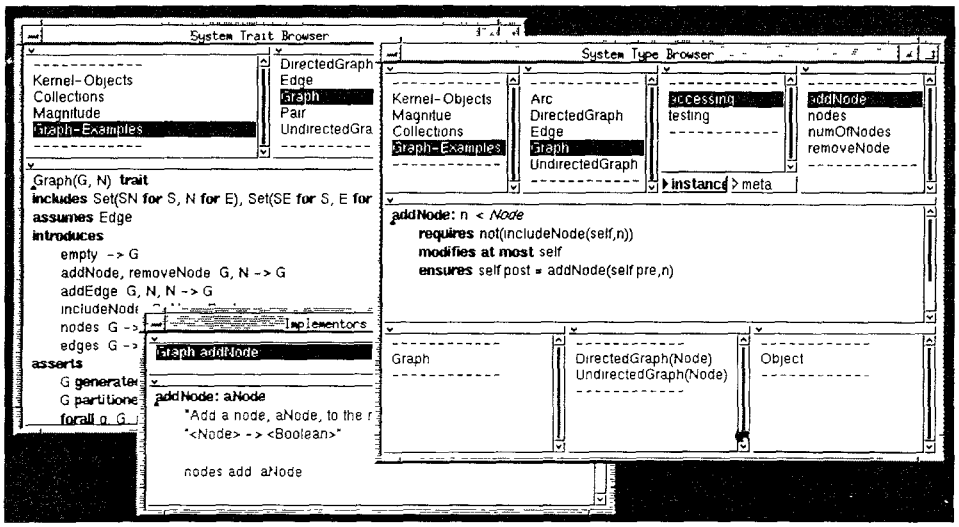


Fig. 1. Larch/Smalltalk specification browsers: a trait browser (left) and a type browser (right). Also shown is a code (method) browser showing an implementation of the currently browsed method specification.

subtyping and inheritance of specification in Larch/Smalltalk. In Section 5, simple type specifications are extended to describe parameterized types. In Section 6, we show several example specifications to give readers some flavor of our specification language. We close with a discussion and some concluding remarks.

2. THE LARCH APPROACH TO INTERFACE SPECIFICATION

The Larch family of specification languages [Guttag et al. 1985; Guttag and Horning 1993] is related to both model-oriented specification and algebraic specification. In this style, the specification of underlying abstractions is separated from the specification of state transformations. Thus, a specification of each program module consists of two components. The state transformations of the program, called the *interface components*, are specified in predicate logic using pre- and postconditions, and describe the effect of operation executions on a program state (e.g., changing an object's value or creating a new object). The interface specification provides the information necessary to use the specified module and to write programs that implement it. The underlying abstractions, called the *shared components*, are specified in an equational (algebraic) style, and describe intrinsic properties that are independent of the model of computation (e.g., a set is an unordered collection of elements without duplicates). The idea is to make the interface language dependent on a specific target programming language, and keep the shared language independent of any programming language. The interface components are specified in programming-language-specific Larch interface languages [Wing 1987; Guttag and Horning 1991; Cheon 1991; Jones 1991;

Leavens and Cheon 1992], and the shared components are written in the Larch Shared Language (LSL) [Guttag and Horning 1993, chap. 4].

The interface specifications are model oriented while the shared components are equational. In the Larch family, there is a clear distinction between the specification of abstract models and the specification of interfaces of program modules. Thus, an interface specification cannot be used to build abstract values of another module, which implies that it also cannot be used to write pre- and postconditions of another interface specification. This is allowed in model-oriented specification languages like Z and VDM because they do not specify language-specific interfaces. On the other hand, the vocabulary for Larch interface specifications can be arbitrarily enriched since it comes from the user-written shared components. Larch provides a set of shared components (traits) in the form of LSL Handbook [Guttag and Horning 1993, appendix A].

Shown below is an interface specification of a method `remove:` of the Smalltalk class Set. The shared component, trait Set, is shown in Figure 2.[1]

`remove:` e <: Elem
returns e1 <: Elem
requires e ∈ $self_{pre}$
modifies self
ensures $self_{post}$ = delete($self_{pre}$, e) ∧ e1 = e

The method takes an object of type Elem, denoted by e, and returns an object of the same type, denoted by e1. The name "self" denotes the receiver, i.e., the object to which the specified message is sent, and "$self_{pre}$" and "$self_{post}$" denote the values of the receiver just before and after the method invocation. The precondition in the **requires** clause says that e must be an element of the receiver; that is, clients are assumed to invoke the method within an element of the receiver. The postcondition in the **ensures** clause asserts that the value of the receiver after method evaluation is the same as (=) that of the receiver before method invocation with the argument e deleted, and the value of the returned object is the same as the value of the argument. The **modifies** clause asserts that the method may mutate only the receiver, and nothing else. The precondition constrains the clients while the **modifies** clause and the postcondition constrain the implementors. The operators appearing in the **requires** and **ensures** clauses (e.g., ∈ , = , and delete) are defined precisely in the shared component (the trait Set).

Figure 2 shows the specification of the shared component, the trait Set, which describes a mathematical notion of finite sets. The following is mainly a summarization of Guttag and Horning [1993, chap. 4]. A trait specifies a mathematical model for interface specifications and describes intrinsic properties that are independent of the model of computation; that is, there is no concept of state, mutation, storage, etc. A trait is an equational specification

[1]The connection between the interface component and the shared component is not shown here. How this connection is made is discussed in the following section.

```
Set(E,S): trait
includes Integer
introduces
    {}:→ S
    insert: S, E → S
    delete: S, E → S
    __ ∈__: S, E → Bool
    isEmpty: S → Bool
    size: S → Int
asserts
    S generated by {}, insert
    S partitioned by isEmpty, ∈
    forall s: S, e, e1: E
        delete({},e) == {}
        delete(insert(s,e),e1) == if e = e1 then delete(s,e1) else insert(delete(s,e1),e)
        ¬(e ∈ {})
        e ∈ insert(s,e1) == if e = e1 then true else e ∈ s
        size({}) == 0
        size(insert(s,e)) == if e ∈ s then size(s) else size(e) + 1
        isEmpty(s) == size(s) = 0
```

Fig. 2. A trait set specified in LSL.

with some additional constructs. It consists of two parts: operator declarations and assertions. A set of *operators* with their signatures is introduced first, and is followed by a set of assertions after the keyword **asserts**. The assertion part specifies a set of constraints on the operators by means of equations and other clauses. An equation consists of two terms of the same sort, separated by = = ; the third equation is an abbreviation for $\neg(e \in \{\})$ = = true.

A trait denotes a *theory* in typed first-order logic with equality. Each theory contains the trait's assertions, the conventional axioms of first-order logic, everything that follows from them, and nothing else. A theory can be strengthened by some additional constructs. A **generated by** clause adds an inductive inference rule to trait's theory. For example, saying that sort S is generated by "{}" and "insert," asserts that each term of sort 'S" is equal to a ground term whose only operators are "{}" and "insert." A **partitioned by** clause asserts that all distinct values of a sort can be distinguished by a given list of operators; this adds a deductive inference rule to the theory. For example, "insert(insert({}, 0), 1)" and "insert(insert({}, 1), 0)" denote the same value, i.e., the set (in a mathematical sense) with two elements "0" and "1."

The **includes** clause in the second line says all of the trait Integer [Guttag and Horning 1993, Appendix A] is made part of the trait Set; that is, the trait Set simply adds trait functions, axioms, etc. to those in the trait Integer. This is one way of combining traits in LSL. For example, the signature and the meaning of "+" comes from the included trait Integer. Boolean operators (true, false, $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$) and some heavily overloaded operators (if-

then-else, = , ¬ =) are built into LSL; in other words, traits defining these operators are implicitly included in every trait.

3. SIMPLE SPECIFICATIONS IN LARCH / SMALLTALK

A Larch/Smalltalk type is an abstraction of a set of Smalltalk classes. As a class is the unit of modularity in Smalltalk, a type is the unit of modularity in Larch/Smalltalk. There are several reasons for specifying in terms of types, rather than of Smalltalk classes:

—A Smalltalk class is a unit of implementation, rather than a unit of behavioral abstraction. As a result, an abstraction is often spread across classes. For example, Booleans are implemented by three classes: Boolean, True, and False. However, it is more intuitive to specify them as one type, say Boolean.

—A class inherits implementations, not specifications. A superclass may specify that subclasses must define a method that a particular subclass does not define, or a subclass can redefine a method to make it inaccessible. We want specification inheritance to be based on behavior (subtyping), not implementation (subclassing) [Cook 1989; LaLonde 1989; America 1991].

—We want multiple inheritance of specifications; that is, we want a type to be a subtype of more than one type. However, classes in Smalltalk can have only one superclass.

—Smalltalk classes are typically organized to give a high degree of code sharing, not according to their logical relationships. We want to structure our specifications based on their conceptual relationships (subtyping), as opposed to the implementation relationships (subclassing). Clients find it much easier to understand and remember relationships that are logical than those that are side effects of particular implementation decisions [LaLonde et al. 1986; LaLonde 1989].

There are two representations for Larch/Smalltalk specifications. In the Larch/Smalltalk browser, specifications are not just plain text, but organized material. A user writes a specification by editing templates given by the browsers. Two templates are provided for interface specifications: one for type specifications (the header part) and the other for method specifications. After creating a type by filling in the type template in the browser, the user can add, modify, and remove its method specifications as he or she typically does with Smalltalk code browsers to browse classes and their methods. Because the graphical interaction with a browser cannot be shown on paper, we use a textual representation for Larch/Smalltalk specifications in this paper.

Figure 3 shows a specification of type IntegerSet[2] in our textual representation. The type IntegerSet models sets whose elements are integers. Syntactically the specification consists of two parts: (1) a header giving the name of

[2] We use a type name to denote both the specified type and specification itself. This is also true for a method name (method selector). The context should tell clearly whether we mean a type or its specification.

type

type IntegerSet
  trait Set (IntegerSet **for** S, Integer **for** E)

meta methods

`new`
    returns s <: IntegerSet
    ensures s_{post} = {} ∧ **fresh**(s)

instance methods

`insert`: n <: Integer
    modifies self
    ensures $self_{post}$ = insert($self_{pre}$,n)

`remove`: n <: Integer
    requires n ∈ $self_{pre}$
    modifies self
    ensures $self_{post}$ = delete($self_{pre}$,n)

`includes`: n <: Integer
    returns b <: Boolean
    ensures b = n ∈ $self_{pre}$

`size`
    returns n <: Integer
    ensures n = size($self_{pre}$)

`isEmpty`
    returns b <: Boolean
    ensures b = isEmpty($self_{pre}$)

Fig. 3 A Larch/Smalltalk specification of type IntegerSet Note that some identifiers (insert, size, and isEmpty) are overloaded to refer to both method selectors and LSL operators. Since there is no syntactic context in which both can appear, there is no ambiguity. However, to make the distinction absolutely clear, we shall adopt the convention of writing selector names in a `typewriter` font.

the specified type and a link that connects the Smalltalk world and the LSL (mathematical) world and (2) a body consisting of a set of method specifications. The header part is separated from the body by a horizontal line in our textual representation. In the body, two kinds of methods are specified: *instance methods* and *metamethods*. Metamethod specifications and instance method specifications are separated by a horizontal line in the textual representation, and the metamethod specifications precede instance method specifications. In IntegerSet, all the method specifications are instance methods except for the method `new`. An instance method defines a message that is sent to an instance of the specified type. A metamethod specification defines a

message that is sent to an instance of the specified type's metatype, i.e., to an object that represents the type itself, not instances of the type. A metatype corresponds to Smalltalk's metaclass [Goldberg and Robson 1983]. A metamethod typically specifies how to create an instance of the specified type. In the specification browsers, a method specification is classified as an instance or as meta when it is entered to the system by making an appropriate selection with the mouse.

3.1 The Header Part

The header of a type specification establishes a connection to its shared component, called the *used trait*. After the keyword **trait** the name of used trait is given, which is followed by a *type-to-sort mapping* in parentheses (see Figure 3). The type-to-sort mapping, which maps type names in the interface specification to sort names in the used **trait**, identifies the set of abstract values for each type in the specification. The abstract values of a type are the equivalence classes of the algebraic terms of the corresponding sort. For example, the used trait of type IntegerSet is the trait Set (see Figure 2); the type-to-sort mapping says that the type IntegerSet is based on the sort S, and the type Integer is mapped to the sort E. Thus, the abstract values of IntegerSet are the equivalence classes of the terms {}, insert({}, 0), insert(insert({}, 0), 1), and so on—terms of sort S in the trait Set.

The abstract values specified in the used trait are purely mathematical. The domain of abstract values of a type can be restricted in the interface level to a subset of the values defined by the used trait. This may be needed for several reasons, e.g., to reuse existing traits or to cope with anticipated implementation limits and restrictions. The **invariant** clause introduces a predicate that must be preserved by all methods of the specified type. In other words, the invariant restricts the abstract values of objects. It must hold in the initial state (just after creation) of an object and must be left invariant by each method. That is, the invariant must be true of the object's abstract value both before and after invoking any methods. Consequently, an invariant will hold in all *visible* states that can be reached from an initial state by means of message sending; it need not hold in all states, since it might be violated temporarily during method evaluation. For example, if we add

invariant size(self) > 0

to the specification of type IntegerSet, only nonempty sets would be abstract values of the type IntegerSet; i.e., {} would not be a legal value for objects of IntegerSet, though it is a term of sort S. (One would also need to rewrite the specification of the methods `new` and `remove:` to preserve the invariant.) In the invariant predicate, "self," which is short for "self$_{any}$" (see Section 3.2 and Appendix B), denotes the abstract value of an object of type IntegerSet. We use "self" so that the invariant can be thought of as being implicitly conjoined to the pre- and postconditions of all method specifications. If no invariant is specified, "true" is assumed by default.

An object whose abstract value can change from state to state is said to be *mutable*; one whose state cannot change is said to be *immutable*. A type is *mutable* if some of its objects are mutable; otherwise it is *immutable*. For example, integers and Booleans are immutable objects while integer sets are mutable objects. So the types Integer and Boolean are immutable while the type IntegerSet, as specified in Figure 3, is mutable. In Larch/Smalltalk, a type is asserted to be mutable or immutable with a **mutation** clause. If this clause is omitted, the specified type is assumed to be mutable by default. The header part in Figure 3 is thus an abbreviation for

type IntegerSet
  mutation true
  trait Set (IntegerSet **for** S, Integer **for** E)

3.2 Method Specifications

A method specification defines a message that can be successfully sent to the objects of the specified type (or metatype in the case of metamethod specification). All the method specifications together describe the protocol of the type. The behavior of a method is specified by the relationship between the inputs in the initial state and the output (return value) in the final state by pre- and postconditions [Hoare 1969]. As an example, consider the method `includes:`.

`includes:` n <: Integer
  returns b <: Boolean
  requires true
  ensures b = n ∈ $self_{pre}$

The method takes an integer and returns a Boolean. Since the precondition holds trivially, the method can be invoked in any state. The postcondition asserts that "true" is returned if n is an element of $self_{pre}$; otherwise, "false" is returned. The notation "$self_{pre}$" means the value of the receiver (i.e., the object to which the message **includes**: is sent) just before the method invocation. The meaning of the LSL operator ∈ is defined in the used trait.

Syntactically, a method specification consists of two parts: the header and the body. The header provides the information necessary to invoke the specified method while the body describes the behavior of the method, i.e., the effect of sending a message that invokes the specified method. The header is similar to that of Smalltalk methods except that we decorate the input arguments with their types, and optionally name the returned object and specify its type. If the **returns** clause is omitted, the receiver is assumed to be returned by default. The body consists of a pair of assertions in the first-order predicate calculus: a **requires** clause and an **ensures** clause. A **requires** clause specifies the precondition that must hold to invoke the specified method. If the precondition is not satisfied, nothing is guaranteed. An omitted **requires** clause is interpreted as equivalent to "**requires** true"; i.e., the method can be invoked in any state. An **ensures** clause states the postcondition that the specified method must establish upon termination; i.e., the postcondition is guaranteed to hold when the method evaluation is

completed. The precondition constrains the clients while the postcondition constrains the implementors.

The semantics of a method specification is a *total correctness* semantics; that is, a method satisfies a method specification if, when it is invoked in a state in which the precondition holds, the method evaluation terminates. In general, a method specification M specifies a nondeterministic state transformation delivering an object as its result. This is formally modeled by a ternary relation among two states and an object. If an $\langle s_1, s_2, o \rangle$ is an element of that relation, we say that an implementation of M *terminates* in the *initial state* s_1, transforming s_1 to the *final state* s_2, and returning o. If for a given state s_1 there is no state s_2 and object o such that $\langle s_1, s_2, o \rangle$ is an element of the relation, we say that the implementation of M does not terminate in the state s_1. Thus total correctness of an implementation requires that for each state s_1 in which the precondition is satisfied, there must be at least one s_2 and o such that $\langle s_1, s_2, o \rangle$ is an element of the relation. In general, there may be more than one such element in the relation; that is, a method specification need not specify a deterministic method.

The assertions in the pre- and postconditions are stated in the first-order predicate calculus. Boolean connectives ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$), the universal quantifier ($\forall$), and the existential quantifier ($\exists$) can be used to compose an assertion. Identifiers and names that can be used in an assertion are

—the implicit input argument "self," which denotes the receiver of the specified message,

—the names of the formal arguments, the name of the formal result (the returned object),

—locally bound logic variables, e.g., `n` in $\forall$(`n:E`)[`n` $\in$ `s` $\Leftrightarrow$ `n` $\in$ `t`], *and*

—operator names from the used trait (e.g., $\in$, insert, delete, size, etc. in the type IntegerSet).

The terms in assertions must be *sort correct* in the sense that operator applications conform to their signatures specified in the traits [Cheon 1991] (see Section 3.3). This is similar to the notion of type-correctness in programming languages.

In the specification of a method that can mutate its arguments, it is usually necessary to refer to the value of an object in two different states: the states before and after the method invocation. Sometimes it is necessary to refer to the identity of an object, that is, to say, the object itself, not its value. The value of an object in the initial state is called its *initial value*; the value in the final state is called its *final value*. Input arguments (including "self") can be qualified with the subscript *pre*, and both input arguments and the return argument may be qualified with the other value qualifier: the subscript *post*. An argument subscripted with a value qualifier denotes the value in the appropriate state (o_{pre} denotes the initial value o, and o_{post} the final value). Arguments can also be qualified with an object qualifier (subscript *obj*) to denote their object identities. Thus o_{obj} denotes the object o as opposed to its value. Qualifications are often redundant, so we adopt certain defaults depending on the context in which an identifier appears. In both the **requires**

clause and **ensures** clause we usually refer to values, so identifiers are qualified with the subscript *pre* by default; one exception is that in the **ensures** clause an output argument is qualified by *post*. On the other hand, in the **modifies** clause (see below) and in the special predicate **fresh** one refers to objects; hence, identifiers in these contexts are qualified by the object qualifier (*obj*) by default.

In Smalltalk, a method can *mutate* an object; i.e., a method can change the state of an object (for example, by assigning to the object's instance variables). To help reasoning about mutation, we insert an optional **modifies** clause in the body of a method specification. This clause asserts that only those listed objects may be mutated as the result of method invocation. This is a strong indirect assertion that no other objects, except for those listed, are allowed to change their abstract values. An omitted **modified** clause is equivalent to the assertion "**modifies nothing**," meaning no objects are allowed to mutate their values. As an example, consider

```
remove: n < Integer
  requires n ∈ self
  modifies self
  ensures self_post = delete(self, n)
```

The method specification says that `remove:` takes an integer argument, and may mutate the receiver to make its value, in the final state, equal to that of deleting the argument from the initial value of the receiver. Since the `returns` clause is omitted, the receiver ($self_{obj}$) is assumed to be returned by default. The method can change the value of the receiver, but can mutate neither the arguments, nor any other objects. More formally, the meaning of a method specification with a **modifies** clause can be given by the predicate: *requires-clause* ⇒ (*modifies-clause* ∧ *ensures-clause*), which must be satisfied by the relation computed by an implementation. In an immutable type, "self" must not appear in the **modifies** clause.

To specify object creation in the postcondition, a special predicate **fresh** is used. The **fresh** predicate asserts that its arguments are newly created as the result of the method invocation. That is to say, these objects do not exist in the initial state, but do in the final state. If there is no **fresh** predicate in the postcondition, the method is not allowed to create any new objects that are visible in the final state. (Technically, in addition to those listed in the **fresh** predicate, a method may create other new objects in the intermediate states that are not visible in the final state. These are *temporary objects* that exist only for the duration of the method evaluation.) The set of objects accessible in the final state must be a subset of the union of the set of objects in the initial state and all those objects listed in **fresh** clauses. For example, consider a method with selector `union:` which may be specified in the type IntegerSet as:

```
union: s < :IntegerSet
  returns t < :IntegerSet
  "Return the union of s and the receiver."
  ensures fresh(t) ∧ ∀(i:Int)[i ∈ t ⇔ i ∈ s ∨ i ∈ self]
```

[ident] $$\Sigma; H, x : S \vdash x : S,$$

[bool] $$\Sigma; H \vdash \texttt{true} : \texttt{Bool},\ \Sigma; H \vdash \texttt{false} : \texttt{Bool}$$

[neg] $$\frac{\Sigma; H \vdash E : \texttt{Bool}}{\Sigma; H \vdash \neg E : \texttt{Bool}}$$

[logic] $$\frac{\Sigma; H \vdash E_1 : \texttt{Bool},\ \Sigma; H \vdash E_2 : \texttt{Bool}}{\Sigma; H \vdash E_1 \diamond E_2 : \texttt{Bool}} \quad \text{for } \diamond \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$$

[quant] $$\frac{\Sigma; H, \vec{x} : \vec{S} \vdash E : \texttt{Bool}}{\Sigma; H \vdash \forall(\vec{x}:\vec{S})\,[E] : \texttt{Bool},\ \Sigma; H \vdash \exists(\vec{x}:\vec{S})\,[E] : \texttt{Bool}}$$

[cond] $$\frac{\Sigma; H \vdash E_1 : \texttt{Bool},\ \Sigma; H \vdash E_2 : S,\ \Sigma; H \vdash E_3 : S}{\Sigma; H \vdash \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3 : S}$$

[equal] $$\frac{\Sigma; H \vdash E_1 : T,\ \Sigma; H \vdash E_2 : T}{\Sigma; H \vdash E_1 = E_2 : \texttt{Bool},\ \Sigma; H \vdash E_1 \neg= E_2 : \texttt{Bool}}$$

[opapp] $$\frac{\Sigma; H \vdash \vec{E} : \vec{S},\ \Sigma \vdash f : \vec{S} \rightarrow T}{\Sigma; H \vdash f(\vec{E}) : T}$$

[paren] $$\frac{\Sigma; H \vdash E : S}{\Sigma; H \vdash (E) : S}$$

[quali] $$\frac{\Sigma; H \vdash E : S\_\texttt{Obj}}{\Sigma; H \vdash E_{val} : S,\ \Sigma; H \vdash E_{obj} : S\_\texttt{Obj}} \quad \text{for } {}_{val} \in \{_{pre}, {}_{post}, {}_{any}\}$$

[fresh] $$\frac{\Sigma; H \vdash \vec{E} : \vec{S\_\texttt{Obj}}}{\Sigma; H \vdash \textbf{fresh}(\vec{E}) : \texttt{Bool}}$$

Fig. 4. Sort inference rules for Larch/Smalltalk.

The method takes an integer set and returns another integer set. The result, `t`, is a new set containing only those integers that are elements of either the input argument set `s`, or the receiver. The result of sending the message `union:` to an IntegerSet object would be a newly created set that did not exist in the initial state; i.e., it is not an alias to an existing set object which happens to have the same value.

As shown in the above example, comments are given in specifications by placing them inside a pair of double quotes.

3.3 Sort Checking

This section describes how to check well-formedness of assertions in the pre- and postconditions. Readers may skip this section at their first reading.

Assertions in the pre- and postconditions (also the invariants) must be *sort correct* in the sense that LSL operator applications conform to their signatures specified in the traits [Wing 1983]. Figure 4 shows the Larch/Smalltalk sort inference rules for sort-checking assertions, based on the abstract syntax

for assertions (see Appendix A). An inference rule of the form

$$\frac{h_1, h_2}{c_1, c_2}$$

means that the truth of conclusions c_1 and c_2 follow from the truth of hypotheses h_1 and h_2; that is, to prove c_1 and c_2 one needs to show that both h_1 and h_2 hold. The hypotheses are optional; if omitted (in which case the horizontal line is omitted, too), the rule becomes an axiom.

Sort checking as stated in the inference rules uses both a sort *environment* H and a *signature* Σ. A sort environment H can be thought of as a set of sort assumptions, pairs of identifier, and sort. An assumption of the form $x{:}T$ means that the identifier x has sort T, and $\vec{x}{:}\vec{S}$ means that each x_i has sort S_i. The notation "$H, x{:}T$" means H extended with the assumption $x{:}T$ (where the extension replaces all assumptions about x in H). A signature Σ contains the signature information of all the LSL operators that can appear in assertions of interface specifications. It is a set of LSL operators signatures of the form: $f{:}\ \vec{S} \rightarrow T$ and is obtained by collecting all the operator declarations in all the used traits of the specified type, the argument types, and the return type. Thus Σ is static in the sense that it is fixed while sort-checking a given method specification, but different H's are used to sort-check different subexpressions. The notation "$\Sigma;\ H \vdash E{:}\ T$" means that given the signature Σ and the sort environment H, one can prove that the expression E has sort T using the inference rules; hence E is sort correct.

The first two rules ([ident] and [bool]) are axioms, inference rules without hypotheses. The axiom [ident] says that one can always retrieve information from Σ, and the axiom [bool] asserts that both `true` and `false` always have the built-in sort `Bool`. Most rules are straightforward. For example, the rule [quant] says that if E has sort `Bool` in the signature Σ and the sort environment H extended with assumption $x{:}S$, then the quantified terms $\forall(\vec{x}{:}\vec{S})[E]$ and $\exists(\vec{x}{:}\vec{S})[E]$ are sort correct and have sort `Bool`.

The heart of the inference rules is the rule [opapp], which tells how to sort-check LSL operator applications. If $\vec{E}$ has sort $\vec{S}$ and f has signature $\vec{S} \rightarrow T$, then in the application of f to E, $f(E)$ is sort correct and has sort T. If f is an infix operator, $f(E)$ should be understood appropriately. The notation $\Sigma \vdash f{:}\ \vec{S} \rightarrow T$ means that an LSL operator f with signature $\vec{S} \rightarrow T$ is in the signature Σ; this allows overloading of f with different arguments sorts as in LSL. (There is no subsorting in LSL.)

There are two sorts associated with each type: an *object sort* and a *value sort*. The object sort models the specified type's objects, and the value sort models the abstract values of the objects in a particular state. The introduction of an object sort is needed to treat a contained object as a part of the value of a containing object, i.e., because of object sharing and mutation. Objects are treated as a special kind of value. This is described in Figure 5, which shows the Larch/Smaltalk view of Smalltalk program states. Let Obj be a set of objects, partitioned into subsets according to their types, and let Val be a sort-indexed family of sets of abstract values. The environment

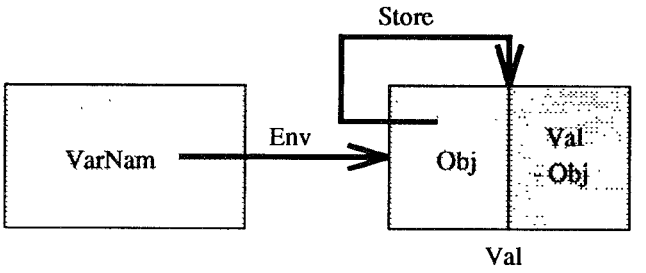


Fig. 5. The Larch/Smalltalk view of program states.

component (Env) of the state maps program variable names (VarNam) to the objects (Obj) denoted by the variables, and store (Store) maps objects to their values. Since objects are also values, the store can map an object (a containing object) to another object (a contained object), which can be mapped to yet another object (a contained object of the contained object) and so on.

The sort of a term denoting the value of an object is a value sort—it can be an object sort because the object can contain another objects (i.e., an object sort is a special kind of value sort). The sort of a term denoting an object itself must be an object sort. If a type T is mapped to a sort S by the type-to-sort mapping, T's object sort is denoted by $S\_\texttt{Obj}$, and T's value sort is denoted by $S\_\texttt{Val}$, which is abbreviated as S. The inference rule [quali] shows the relationship between object sorts and value sorts. If E is a term of sort $S\_\texttt{Obj}$, then a value-qualified E (e.g., E_{pre}) is of sort S. For example, if x is a program variable of type T, and T is mapped to a sort S, then x_{obj} is a term of sort $S\_\texttt{Obj}$ (because x_{obj} denotes the object x), and x_{pre}, x_{post}, and x_{any} are of sort S (because they denote the values of the object x).[3] In the sort inference rules, we assume that terms are fully qualified. Refer to Appendix B for the default qualification rules for self and formal arguments.

4. SPECIFICATION INHERITANCE AND SUBTYPING

Larch/Smalltalk is the first interface specification language that permits inheritance of specifications [Cheon 1991]. Inheritance of specifications permits specifiers to construct specifications incrementally. To specify a type incrementally, we state how it differs from other types, called *supertypes*, by adding additional features; this makes the new type a subtype of other types. Syntactically this is done with the **supertypes** clause in the type header. After the keyword **supertypes**, all the direct supertypes of the specified type are listed; if the **supertypes** clause is omitted, the specified type is assumed to be a direct subtype of the type Object, the ultimate supertype of all types. For example, assuming the existence of type Collection, an abstraction of

[3]The term E_{any} denotes the value of E in some unknown state; this form is used to refer to an object's value in the invariant clause.

collection types such as lists and arrays, the header part of IntegerSet can be respecified as follows:

```
type IntegerSet
  supertypes Collection
  trait Set(IntegerSet for S, Integer for E)
```

The type IntegerSet is a direct subtype of the type Collection, thus inheriting its properties (e.g., method specifications). Aside from inherited methods, only additional methods and changed methods need to be specified in the subtype. Specifying a type in terms of its differences from its supertypes leads to shorter specifications, and such specifications are easier to maintain. To start things off, a large number of type specifications are provided in the system, structured into a hierarchy based on their conceptual relationships [Cook 1992], with the most general type Object at the root. The type Object specifies properties concerned with all objects; it has no method specification.

If type S is specified to be a direct subtype of type T (i.e., T is a direct supertype of S), then S *inherits* the specified properties of T. That is, S inherits the invariant and method specifications of T, if any. A subtype's invariant is the conjunction of all of its supertypes' invariants and the invariant stated explicitly in the subtype with **invariant** clause. An instance method with the same selector specified by more than one supertype in different subtyping chains must be respecialized by the subtype to resolve potential conflicts; that is, the method must be specified in the subtype. An alternative approach to resolving multiple inheritance conflicts would be to disjoin the preconditions, conjoin the postconditions, and intersect the **modifies** clauses of all the conflicting method specifications; this would ensure the behavioral subtyping. However, when the objects listed in the **modifies** clauses of conflicting method specifications differ, such a method specification would usually be unsatisfiable, because the postcondition would require objects to change states that no longer appear in the **modifies** clause.

What does an inherited method specification mean? The basic problem is to ensure that the operators used in the inherited method specification, which were written for abstract values of the supertype, can be applied to the abstract values of the subtype [Leavens 1993]. The simplest and most general answer, adopted by Larch/Smalltalk, is to view inheritance as textual expansion and to require the subtype's used trait to provide a meaning for the operators used in inherited method specifications. That is, the meaning of an inherited method specification is given by reinterpreting the text of the inherited specification with the subtype's used trait. (This technique is also the foundation of specification inheritance in Larch/C++ [Leavens and Cheon 1992].) This technique requires two conditions to be satisfied by the subtype's used trait. Syntactically, the signature of the subtype's used trait must be a superset of that of the supertype's used trait. Semantically, the theory of the subtype's used trait must include that of the supertype's used trait. If some property of the supertype's abstract values was not preserved by the subtype's used trait, such as an operation that was idempotent failing to be so in the subtype's used trait, then one could not correctly reason about

the abstract values of subtype objects using the inherited specifications. Therefore, to allow such reasoning about inherited specifications, the theory of a subtype's used trait should be a consistent extension of the theories of its supertypes' traits. One way to ensure this is to define the trait functions that apply to abstract values of the subtype by a homomorphic coercion function from subtype abstract values to supertype abstract values [Reynolds 1980; Goguen and Meseguer 1987; Bruce and Wegner 1990; America 1991]. The advantage to the more general approach taken in Larch/Smalltalk is that the homomorphic coercion functions can be used whenever possible, but the specifier is not limited to this technique. (For example, one can use homomorphic relations instead of functions.)

Subtype relationships are not only useful in easing specification, they may also be used to aid verification or informal reasoning about programs. To fulfill this role, whenever S is a subtype of T, each object of type S must act like some object of type T, when used from the perspective of T's specification. In the specification terms this means, that for each method M specified both in S and T, (1) the precondition of M in T implies the precondition of M in S and (2) the postcondition of M in S implies the postcondition of M in T. Formal requirements for such behavioral subtyping [Meyer 1988a; Leavens and Weihl 1990; America 1990; Leavens 1991; Liskov and Wing 1993a; 1993b] involve either semantic modeling or theorem proving. The Larch proof assistant LP [Guttag and Horning 1993, chap. 7], because it accepts LSL syntax, could be used to prove such properties. Larch/Smalltalk itself checks only for the traditional syntactic constraints, which we call the *syntactic subtyping rule*. The syntactic subtyping rule says that for each subtype S of a supertype T, if an instance method M is specified in both S and T, the following conditions must hold [Cardelli and Wegner 1985; Schaffert et al. 1986]:

—For every input argument of M except for the implicit argument "self," its type in T must be a subtype of the corresponding type in S.
—The return type of M in S must be a subtype of the return type of M in T.

That is to say, an argument type of a method can only be *generalized* in a subtype, whereas the result type can only be *specialized*. The reversal of direction for arguments is why this rule is called *contravariant*. Contravariance seems a bit awkward in practice, because a programmer typically wants to specialize rather than to generalize arguments. An alternative is to use *covariance*, which means that argument types can also be specialized. Such type systems are not statically sound and are hard to reason about [Cook 1989]. Additionally, contravariance does not seem to cause many problems at the specification level. The syntactic subtyping rule, together with specification inheritance, guarantees that a message understood by objects of a type is also understood by objects of its subtypes. However, the effects of receiving messages are not guaranteed to be the same. Semantic correctness (legal subtyping) is left in the hands of the specifiers.

Set(Elem)

type Set
  parameters Elem
  trait Set (Set(Elem) **for** S, Elem **for** E)

meta methods

new
  returns s <: Set(Elem)
  ensures s_{post} = {} ∧ **fresh**(s)

instance methods

insert e <: Elem
  modifies self
  ensures $self_{post}$ = insert($self_{pre}$,e)
  ⋮

Fig. 6. The parameterized type specification Set.

In Larch/Smalltalk, a subtype does not have to be implemented by a subclass, and a subclass does not have to implement a subtype. This separation of subtyping from subclassing gives a great freedom both in design and implementation. The decoupling of subtyping from subclassing is the feature that most clearly distinguishes Larch/Smalltalk from other object-oriented specification languages.

5. PARAMETERIZED SPECIFICATIONS

5.1 Simple Parameterized Specification

In Section 3 we specified sets whose elements are integers. Of course, integers are not the only element types; there are many applications in which we want to have sets with elements other than integers. We would like to have a single specification that captures all these different kinds of sets. A parameterized type specification provides a simple way to do this. The major idea introduced by parameterized type specifications is that of a *type parameter*. For example, in the specification of Set (see Figure 6), Elem is a type parameter representing the type of element objects. A type parameter is a place holder that is replaced by an actual type later, when the specification is instantiated. It can be used freely in places where a type name is expected.

The parameterized type specification can be viewed as a notational abbreviation from which specifications are generated by supplying a concrete type for the type parameters. For example, supplying Integer to the specification Set produces type Set(Integer), the type of sets whose elements are of type Integer. Similarly, it can be instantiated to Set(Character), Set(String), and so on. All the instantiated specifications will have a similar property, e.g.,

they will have the same set of methods. In itself, Set is not a type (there are no objects of type Set), but rather a type generator in the sense that it can generate types by instantiation.

The introduction of the type parameter Elem makes it possible to specify methods that take arguments or return results of type Elem. That is, for each instantiation the argument or return type will be different, depending on the actual parameter type. For example, in Set(Integer) the `insert:` method takes an integer as its argument, whereas in Set(String) it takes an object of type String.

5.2 Bounded Quantification

The simple parameterized type specification introduced in the previous section cannot make any assumptions about the objects of their type parameters since any type could be used for these parameters. In implementation terms, this means that a parameterized type cannot send any message to an object of its type parameters, because it is not known whether the actual types for the parameters have an appropriate method. In reasoning, this means that we cannot assert anything about the type parameters. In many applications, however, it is useful to have more information about the type parameters, for instance, the presence of certain methods. To help reason about parameterized types, we can combine the idea of type parameters and subtyping into a notion called *bounded quantification* [Cardelli and Wegner 1985]. Each type parameter is bounded by a type. Only subtypes of a given type (upper bound) are allowed in place of type parameters. For example, the header part of specification Set in Figure 6 can be replaced by the following:

type Set
  parameters Elem ≤ ObjectWithEquality
  supertypes ObjectWithEquality
  trait Set (Set(Elem) **for** S, Elem **for** E)

The type parameter Elem is bounded by the type ObjectWithEquality, a direct subtype of the type Object with a specification for the binary method = (equal). Only subtypes of ObjectWithEquality are allowed as the actual types for the parameter. For example, Set(Object) is not well formed because Object is not a subtype of ObjectWithEquality. This restriction to the type parameter is reasonable because the specification of Set assumes that two objects of type Elem can be compared for equality. By default, an unbounded type parameter is bounded by type Object, i.e., any type can be used in place of such a type parameter. Thus, the simple parameterization introduced in the previous section is a special case of bounded quantification in which all the type parameters are bounded by the type Object.

6. AN EXTENDED EXAMPLE

To give some flavor of our specification language, we specify several interface modules in Larch/Smalltalk. The chosen examples are graphs. Mathematically, a *graph* G is an ordered tuple $(V(G), E(G))$, consisting of a set $V(G)$ of

```
GraphTrait(N,G): trait
includes Set(N,SN), Set(E,SE)
G tuple of nodes: SN, edges: SE
E tuple of head: N, tail: N
introduces
    includesNode, isolatedNode: G, N → Bool
asserts
    forall g:G, sn: SN, se: SE, n,m,m1: N
        includesNode(g,n) == n ∈ g.nodes
        isolatedNode([sn,{}],n)
        isolatedNode([sn,(insert([m,m1],se))],n) ==
            ¬(n = m ∨ n = m1) ∧ isolatedNode([sn,se],n)
```

Fig. 7. The trait GraphTrait.

vertices and a set $E(G)$ of *edges*, where an edge is a pair of (not necessarily distinct) vertices of G. The first example of an edge is called the *head*, and the second element is called the *tail*. If the edges are ordered, the graph is *directed*; otherwise it is *undirected*. For directed graphs, we use the term *arcs* instead of edges.

We will specify two types, DirectedGraph and UndirectedGraph, which describe directed graphs and undirected graphs, respectively. To take advantage of specification inheritance, we abstract all the features common to both directed graphs and undirected graphs into an abstract type Graph, and specify the two types to be direct subtypes of the abstract type.

The underlying model for the type Graph is shown in Figure 7. A graph G is a tuple of nodes and edges, where nodes is of sort SN (set of N) and where edges is of sort SE (set of E). We use the term *nodes* instead of vertices in our specification. An edge E is again a tuple of nodes N, whose first and second elements are denoted by head and tail, respectively. The tuple definition is an LSL shorthand notation for introducing fixed-length tuples [Guttag and Horning 1993, chap. 4]. For example, defining "G tuple of nodes: SN, edges: SE" introduces a tuple constructor ([_, _]), observer operators (_.nodes, _.edges), and updating operators (set_nodes and set_edges, both of which produce new tuples) with appropriate axioms.

The trait GraphTrait defines two operators: includesNode and isolatedNode. The operator includesNode tells whether a vertex is in a graph, while isolatedNode tests if a vertex is isolated from others. The operator $\in$ in the axiom for includesNode is the set membership operation, and comes from the included trait Set. The trait Set found in the *LSL Handbook* [Guttag and Horning 1993, Appendix A] defines a mathematical model for finite sets. It is similar to the trait Set in Figure 2 except that it also defines typical set operations, like $\cup$, $\cap$, $-$ (set difference). A vertex is isolated if the graph has no edge at all or if there is no edge between the vertex itself and some other vertex in the graph. The second and third axioms state this.

The trait UndirectedGraph shown in Figure 8 defines an abstract model for the type UndirectedGraph. In addition to properties stated in the included

```
UndirectedGraphTrait(N,G): trait
includes GraphTrait(N,G)
introduces
    includesEdge: G, E → Bool
asserts
    forall g:G, e: E
        includesEdge(g,e) == (e ∈ g.edges) ∨ ([e.tail,e.head] ∈ g.edges)
```

Fig. 8. The trait UndirectedGraphTrait.

```
DirectedGraphTrait(N,G): trait
includes GraphTrait
introduces
    includesArc: G, E → Bool
asserts
    forall g:G, a: E
        includesArc(g,a) == a ∈ g.edges
```

Fig. 9. The trait DirectedGraphTrait.

trait GraphTrait, it defines an operator includesEdge. An edge e is included in an undirected graph g if the edge set of g (g.edges) includes e or [e.tail, e.head]. This is because the edge e has no direction associated with it.

A mathematical model for the type DirectedGraph, the trait DirectedGraphTrait is shown in Figure 9. It is similar to the trait UndirectedGraphTrait except that now each edge has a direction attached to it; thus, the axiom for includesArc is "includesArc(g, a) = = a ∈ edges(g)."

Since we have formal models for all three types, it is time to specify the types at the interface level. Because graphs are useful with a variety of vertices, all the types are parameterized with a type parameter Node, which stands for the type of vertices. Figure 10 shows the abstract type Graph. It is an abstract type in the sense that it does not have any metamethod specifications, that is, no objects of this type can be created. Its sole purpose is to be a common supertype of its two concrete subtypes, which will be specified later.

The **invariant** clause in Figure 10 says that both the head and tail of an edge must be nodes of the graph. That is, the abstract values of Graph are those terms of sort G in the trait GraphTrait (see Figure 7) that satisfy the invariant predicate. For example, [{}, {}] is one possible abstract value, a graph with no nodes and no edges. However, [{n}, {[n, m]}] cannot be an abstract value of a Graph object even though it is a term of sort G; it does not satisfy the invariant.

The type specifies five instance methods: `addNode:`, `removeNode:`, `chooseNode`, `nodes`, and `numOfNodes`. Terms in the pre- and postconditions of these method specifications come from the trait GraphTrait.

Given a vertex, not included in a graph, the method `addNode:` adds the node to the graph. The postcondition says that $self_{post}$ (final value of self) is equal to self (initial value of self) with its vertices replaced by the union of self nodes (vertices of self in the initial state) and the vertex to be added.

Graph(Node)

type Graph
  parameters Node ≤ ObjectWithEquality
  trait GraphTrait (Graph(Node) **for** G, Set(Node) **for** SN, Node **for** N)
  invariant ∀(e:E)[e ∈ self.edges ⇒ ((e.head ∈ self.nodes) ∧ (e.tail ∈ self.nodes))]

instance methods

`addNode`: n <: Node
  requires ¬includesNode(self,n)
  modifies self
  ensures $self_{post}$ = set_nodes(self, (self.nodes ∪ {n}))

`removeNode`: n <: Node
  requires includesNode(self,n) ∧ isolatedNode(self,n)
  modifies self
  ensures $self_{post}$ = set_nodes(self, (self.nodes − {n}))

`chooseNode`
  returns n <: Node
  requires ¬isEmpty(self.nodes)
  ensures includesNode(self,n)

`nodes`
  returns s <: Set(Node)
  ensures **fresh**(s) ∧ s = self.nodes

`numOfNodes`
  returns n <: Integer
  ensures n = size(self.nodes)

Fig. 10. The parameterized specification Graph.

Since there is no **returns** clause, $self_{obj}$ is returned by default. The method `removeNode`: deletes an existing vertex from the graph. The precondition says it can be invoked only with a vertex with no edges associated with it, i.e., the vertex must be isolated. As in the method `addNode`:, $self_{obj}$ is returned by default. The method `chooseNode` is interesting in that its postcondition is underspecified, that is, the specification permits nondeterministic implementation. All the specification says is that the return object is a vertex of self. It does not say which one should be returned if there is more than one vertex. The implementation may use this freedom to improve efficiency. The method `nodes` returns a new set containing all the vertices of self. The method `size` returns the number of vertices in the graph. Note that no method is concerned with edges because it is not known et whether the edges have directions associated with them or not. These are properties to be specified by concrete subtypes.

DirectedGraph(Node)

type DirectedGraph
  supertypes Graph(Node)
  parameters Node ≤ ObjectWithEquality
  trait DirectedGraphTrait (DirectedGraph(Node) **for** G, Set(Node) **for** SN,
      Node **for** N)

meta methods

`new`
  returns g <: DirectedGraph(Node)
  ensures g = [{},{}] ∧ **fresh**(g)

instance methods

`addArcFrom:` n <: Node `to:` m <: Node
  requires includesNode(self,n) ∧ includesNode(self,m) ∧ ¬includesArc([n,m])
  modifies self
  ensures $self_{post}$ = set_edges(self, self.edges ∪ {[n,m]})

`removeArcFrom:` n <: Node `to:` m <: Node
  requires includesArc(self,n,m)
  modifies self
  ensures $self_{post}$ = set_edges(self, self.edges - {[n,m]})

`adjacentNodesFrom:` n <: Node
  returns s <: Set(Node)
  requires includesNode(self,n)
  ensures fresh(s) ∧ ∀(m:N)[m ∈ s ⇔ [n,m] ∈ self.edges]

`adjacentNodesTo:` n <: Node
  returns s <: Set(Node)
  requires includesNode(self,n)
  ensures fresh(s) ∧ ∀(m:N)[m ∈ s ⇔ [m,n] ∈ self.edges]

`adjacentNodes:` n <: Node
  returns s <: Set(Node)
  requires includesNode(self,n)
  ensures fresh(s) ∧ ∀(m:N)[m ∈ s ⇔ [n,m] ∈ self.edges ∨ [m,n] ∈ self.edges]

Fig. 11. The interface specification DirectedGraph.

Figure 11 shows the specification for type DirectedGraph. The type DirectedGraph is parameterized and specified to be a direct subtype of type Graph (see also Section 6.1).

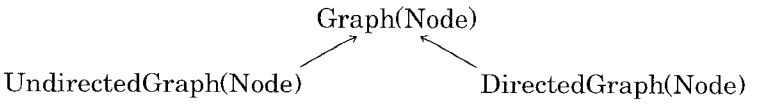
The type DirectedGraph specifies a metamethod `new` which returns an empty directed graph [{}, {}]. Because DirectedGraph is a subtype of Graph, it inherits the invariant and all the method specifications of Graph. In addition

to inherited methods, DirectedGraph specifies five new instance methods: addArcFrom:to:, removeArcFrom:to:, adjacentNodesFrom:, adjacentNodesTo:, and adjacentNodes:. The method addArcFrom:to: inserts a new arc, denoted by a pair of vertices, whereas removeArcFrom:to: deletes an existing arc from the graph. The precondition of addNodeFrom:to: requires that both the head and tail of the arc should be vertices of the graph. Since addNodeFrom:to: is the only method that adds arcs, every object of type DirectedGraph satisfies the invariant inherited from Graph. By our convention, both addArcFrom:to: and removeArcFrom:to: return the object $self_{obj}$. The method adjacentNodesFrom: returns a new set containing all the vertices adjacent *from* a given vertex, while adjacentNodesTo: returns a new set containing all the vertices adjacent *to* a given vertex. The method adjacentNodes: returns a new set of all the vertices adjacent to and from a given vertex.

The type UndirectedGraph, another subtype of Graph, is shown in Figure 12. Its invariant is inherited from Graph. The metamethod new returns an empty undirected graph [{}, {}]. The instance method addEdgeBetween:and: inserts a new edge to the receiver, whereas removeEdgeBetween:and: deletes an existing edge from the receiver. The postcondition of removeEdgeBetween:and: states that it deletes both the edges [n, m] and [m, n]. Because there is no direction associated with an edge, both denote the same edge, an edge between vertices n and m. Both methods return the object $self_{obj}$ by default. The method adjacentNodes: returns a new set containing all the adjacent vertices of a given vertex.

6.1 Subtyping in Parameterized Type Specifications

Both DirectedGraph and UndirectedGraph are specified to be subtypes of Graph. But in a strict sense, neither is a subtype of Graph. In fact, none of the three are types by themselves; rather they are type generators. What we mean by "DirectedGraph is a subtype of Graph" that for each type Node, DirectedGraph(Node) is a subtype of Graph(Node). For example, DirectedGraph(Integer) is a subtype of Graph(Integer), and DirectedGraph(Character) is a subtype of Graph(Character). However, DirectedGraph(Integer) is not a subtype of Graph(Character) nor the other way around. Thus, for three specifications, we have the following subtype hierarchy:



Suppose that the type SmallInteger is a subtype of type Integer. An interesting question is whether DirectedGraph(SmallInteger) is a subtype of Graph(Integer) or vice versa. Let us assume that DirectedGraph(SmallInteger) is a subtype of Graph(Integer). The method addNode: would then require an argument of type SmallInteger in the subtype and an argument of type Integer in the supertype. But this would contradict the syntactic subtype rule, which requires that argument types can only be generalized in subtypes.

UndirectedGraph(Node)

type UndirectedGraph
  parameters Node ≤ ObjectWithEquality
  supertypes Graph(Node)
  trait UndirectedGraphTrait (UndirectedGraph(Node) **for** G, Set(Node) **for** SN, Node **for** N)

meta methods

new
  returns g <: UndirectedGraph(Node)
  ensures g = [{},{}] ∧ **fresh**(g)

instance methods

addEdgeBetween: n <: Node and: m <: Node
  requires includesNode(self,n) ∧ includesNode(self,m) ∧ ¬includesEdge([n,m])
  modifies self
  ensures $self_{post}$ = set_edges(self, self.edges ∪ {[n,m]})

removeEdgeBetween: n <: Node and: m <: Node
  requires includesEdge(self,n,m)
  modifies self
  ensures $self_{post}$ = set_edges(self, self.edges - ({[n,m]} ∪ {[m,n]}))

adjacentNodes: n <: Node
  returns s <: Set(Node)
  requires includesNode(self,n)
  ensures fresh(s) ∧ ∀(m:N)[m ∈ s ⇔ [n,m] ∈ self.edges ∨ [m,n] ∈ self.edges]

Fig. 12. The interface specification UndirectedGraph.

Thus, DirectedGraph(SmallInteger) cannot be a subtype of Graph(Integer). Assuming that Graph(Integer) is a subtype of DirectedGraph(SmallInteger) would lead to a similar conflict with the syntactic subtyping rule. For example, the method chooseNode: returns an object of type Integer in Graph(Integer), and returns an object of type SmallInteger in DirectedGraph(SmallInteger). So, DirectGraph(Integer) cannot be a subtype of Graph(SmallInteger) because it violates the second condition of the subtyping rule saying that the result type can only be specialized in subtypes. Therefore, in general, if S is a subtype of T, then DirectedGraph(S) is not a subtype of Graph(T) and vice versa.

Let us consider subtyping relationships between different instantiations of the same parameterized type specification. Consider the case when we substitute two parameters that are subtypes of each other. For example, is the type Graph(SmallInteger) a subtype of the type Graph(Integer)? The method addNode: requires an argument of type SmallInteger in the first type and the argument of type Integer in the second type. Therefore, for the same

reason as above, Graph(SmallInteger) is not a subtype of Graph(Integer). We can show easily that the subtyping relationship the other way around also conflicts with our subtyping rule. Therefore, in general we do not have a subtype relationship between different instantiations of the same parameterized type, though there are some cases where such a relationship holds [Cook 1989].

7. DISCUSSION

7.1 Related Work

Recently a lot of effort has been put into applying object-oriented concepts to formal specification and reasoning techniques, that is to say, into designing object-oriented specification languages and into specifying and verifying programs in object-oriented programming languages. This effort can be divided into two categories: designing new specification languages and extending existing specification languages with object-oriented concepts. Object orientation is reflected in the specification language ABEL [Dahl 1987] in a class-like construct which defines objects in the conventional imperative sense. ABEL contains mechanisms for constructive and nonconstructive specifications as well as applicative and imperative programming. In GSBL [Clerici and Orejas 1988], an algebraic specification language, one can see full-fledged notions of objects, classes, and inheritance. In the database community, the Oblog$^+$-language [Jungclaus et al. 1991] incorporates object orientation to specify information systems, especially for the conceptual modeling of systems.

Several object-oriented extensions have been proposed for the specification language Z [Hayes 1987] due to its style (e.g., graphical layout of specifications, use of set-theoretic and logical notations, and conventions for decorating input and output variables, etc.) and to its growing use in industry. Schuman and Pitt [1986] described a semantics to accommodate object orientation based on events and histories, though they did not provide the class as a single syntactic construct. Object-Z [Carrington et al. 1989] introduces classes to encapsulate the description of an object's state with its related operations. Complex specifications are then constructed through class inheritance and instantiation. Its class model is also based on the idea of history, which captures the sequence of operations and state changes undergone by an instance (object) of the class. The OOZE System [Alencar and Goguen 1991], based on Z and OBJ3, provides a powerful parameterization mechanism (modules, theories, views) as well as notions of objects, classes, and inheritance. Object orientation was also attempted for the specification languages VDM [Bear 1988] and LOTOS [Mayer 1988]. In Fresco [Wills 1992], a programming environment for developing object-oriented software from specifications based on VDM, a class describes a specification, an implementation, or a mixture of the two. A class is specified with model variables, invariants, and operation specifications. The state of an object is captured by these model variables, i.e., it is a composition of the values of these variables. This

composition of model variables corresponds to the abstract value of an object in Larch/Smalltalk, which in Larch/Smalltalk is specified in LSL. Fresco also distinguishes between the class hierarchy of implementations and the type hierarchy of conformance. But the preference in Fresco seems to be to combine the two in *conformant inheritance*, in which the subclass also happens to implement a subtype [Meyer 1988b].

Larch/Smalltalk is the first Larch interface specification language with subtyping and specification inheritance [Cheon 1991]. Other Larch interface languages with similar features are LM3 (Larch/Modula-3) [Guttag and Horning 1993, chap. 6] and Larch/C++ [Leavens and Cheon 1992; Cheon and Leavens 1993]. Both allow reuse of specifications in the interface level through specification inheritance. The most distinguishing features of Larch/Smalltalk compared to LM3 and Larch/C++ are its simplicity and flexibility, and the separation of types from classes. The syntax and semantics of Larch/Smalltalk are much simpler than LM3 and Larch/C++, partly due to the simplicity of Smalltalk.

The most interesting feature of Larch/C++ is that a class specification can have multiple interfaces: the *public interface* for clients, the *protected interface* for subclasses, and the *private interface* for implementors and friends. This is a very useful feature both in programming and specification. It can be somewhat simulated in Larch/Smalltalk by the disciplined and stylized use of message categories. For example, methods can be categorized depending on whether they are public, protected, or private; in fact, this is what a sensible Smalltalk programmer does with Smalltalk methods. However, this cannot prevent clients from accessing protected or private methods if they want to.

In LM3 [Jones 1991], one can specify a higher-order procedure, a procedure that takes other procedures as its arguments. Similar features are also found in Larch/CLU [Wing 1983] and LCL (Larch/C), [Tan 1992]. The interface (arguments and their types) and the behavior (using pre- and postconditions) of an argument procedure are specified in the header part of the procedure, which takes it as an argument. And a special notation is provided to refer to the pre- and postconditions of the argument procedure in the pre- and postconditions of the higher-order procedure. A similar approach might be taken to specify Smalltalk blocks. LM3 also has support for specifying *threads*, lightweight units of concurrency in Modula-3. A nonatomic routine is specified as sequence of *atomic actions* [Wing 1990]. Concurrency issues are not addressed in Larch/Smalltalk.

7.2 Future Work

7.2.1 *Language Extension.* In Smalltalk, methods can take or return blocks. That is, methods can be higher order. A block is a closure; it contains a parameterized code and an environment. Since Smalltalk control structures such as `ifTrue:ifFalse:` and `whileTrue:` are based on blocks, they are an essential feature of the Smalltalk system. Several approaches to specifying blocks are being examined: (1) modeling them explicitly as state transition in

LSL, (2) specifying in the interface the weakest pre- and postconditions that the argument blocks have to satisfy [Ernst et al. 1982; Jones 1991], (3) using *free functions* as proposed in LCL [Tan 1992], and (4) introducing new predicate operators that can model (repeated) invocation of argument blocks. An interesting fact about blocks is that they allow nonlocal exits; blocks are continuations. A block can exit to the place where it was defined (which may be different from where it was invoked). Since this feature is heavily used by Smalltalk programmers to handle error cases, etc., it should be properly addressed in extending Larch/Smalltalk for specifying block arguments.

Another desirable extension to the current syntax is for specifying exceptions. Smalltalk exception-handling mechanisms are based on the *multilevel resumption model*; an exception can be propagated to multiple levels, and control can later be resumed by the exception-raising module. There are some provisions in Larch interface languages such as Larch/CLU, LM3, and Larch/C ++ for specifying exceptions, but all of them are for the *simple termination model*; an exception is propagated only to the invoking module and control cannot be resumed by the exception-raising module.

Smalltalk allows programming at the metalevel in the sense that classes themselves are represented by objects, called *class objects*. We can refer to these class objects in instance methods, and we can define methods for the class objects, which are called class methods. Classes defining class objects are called *metaclasses*. To specify such class objects, we could specify metatypes much as we specify types. The main problem is to connect the specification of a type with the specification of its metatype, because in Larch/Smalltalk a type may be implemented by more than one class. One idea is to use a notation such as self_{meta} to refer to the receiver's class object in the specification of instance methods; this would allow the class object to be discussed without explicitly naming a particular class object.

7.2.2 *Formal Semantics.* Defining a formal semantics will be the main focus of our future research in Larch/Smalltalk. Informally, a Larch/Smalltalk specification denotes a set of Smalltalk program modules whose interfaces and behaviors conform to the specification. In this context a program module means a class or several classes collectively. One approach to giving a formal semantics would be to define: (1) a common basis (some mathematical notations) between Larch/Smalltalk and Smalltalk and (2) two translation functions, one for specifications and the other for programs. The meaning of a specification would be all the Smalltalk modules whose meaning is implied by the mathematical term to which the specification is translated. For example, let S be a Larch/Smalltalk specification, P be a Smalltalk program module, and T_s and T_p be translation functions. Then the meaning of S, $M[[S]]$ could be:

$$M[[S]] \stackrel{\text{def}}{=} \{P | T_s[[S]] \Rightarrow T_p[[P]]\}.$$

7.2.3 *Verification and Reasoning.* We would like to explore how to use Larch/Smalltalk as a formal basis for verifying and reasoning about Smalltalk programs. Basically we would like to design a Hoare-style proof

logic adapted to object-oriented programming, something like the one discussed in Leavens and Weihl [1990] and Leavens [1991].

7.3 Summary

Behavioral specification of reusable components is more necessary in object-oriented programming than in a conventional programming environment. The lack of such description techniques for Smalltalk has caused poor reuse of its huge library classes and made it hard for programmers to exchange code for possible reuse. Larch/Smalltalk answers these needs with a formal specification language specifically tailored to Smalltalk. Larch/Smalltalk is a Larch interface specification language with notions of subtyping and specification inheritance. One can describe precisely both the behavior and the interface of Smalltalk modules (classes and methods).

The main contribution of this paper is its separation of types from classes. Type is the unit of abstraction for specification. This is an interesting way of introducing a type system (at the specification level) when the underlying language is untyped, and provides natural mechanisms for specifying Smalltalk interfaces. Subtyping allows specifications to be organized according to their conceptual relationships, i.e., in subtype hierarchies, as opposed to implementation relationships. Additionally, specifications can be reused at the interface level by specification inheritance. Parameterization is also allowed to specify a set of related types.

We expect ordinary Smalltalk programmers to learn and use Larch/Smalltalk easily and productively in programming. The flexibility of Larch/Smalltalk is obtained by decoupling the specification unit (type) from the implementation unit (class). Thus, a Larch/Smalltalk can be implemented by a single Smalltalk class, several classes forming a subhierarchy in the subclassing hierarchy, or a set of classes. Also, a type may have several different implementations in a program [LaLonde et al. 1986]. The separation of types from classes gives a great freedom in design and implementation.

To allow specifications to be used practically in the programming process, Larch/Smalltalk specification browsers integrated in the Smalltalk systems were implemented. A preliminary version is available by anonymous ftp from ftp.cs.iastate.edu.

APPENDIX

A. REFERENCE GRAMMAR

This section lists the reference grammar of Larch/Smalltalk in an extended BNF with conventions: (1) nonterminal symbols are enclosed in angle brackets (e.g., ⟨method-header⟩), (2) keywords are written in **bold** face (e.g., **requires**), (3) reserved words and other terminal symbols are written in a `typewriter` font if possible (e.g., `self`, `[ , ]`); otherwise they will be written normally (e.g., ∀, ⇒), (4) optional symbols are surrounded by square brackets (e.g., [**returns**⟨formal-declaration⟩]), and (5) the notation "..."

means that preceding symbol (or a group of optional symbols) can be repeated zero or more times (e.g., ⟨method-specification⟩ ...).

The lexical conventions are the same as those of Smalltalk [Goldberg and Robson 1983]. For example, ⟨identifier⟩ is an arbitrary long sequence of letters and digits whose first character is a letter.

A.1 Type Specifications

⟨type-specification⟩ → ⟨type-header⟩⟨type-body⟩

⟨type-header⟩ → **type**⟨identifier⟩[⟨parameters-clause⟩][⟨supertypes-clause⟩][⟨mutation-clause⟩][⟨uses-clause⟩][⟨invariant-clause⟩]

⟨parameters-clause⟩ → **parameters**⟨type-parameter⟩ [,⟨type-parameter⟩]...

⟨type-parameter⟩ → ⟨identifier⟩[≤ ⟨type-name⟩]

⟨type-name⟩ → ⟨identifier⟩|⟨identifier⟩(⟨type-name⟩[,⟨type-name⟩]...)

⟨mutation-clause⟩ → **mutation**⟨boolean⟩

⟨uses-clause⟩ → **trait**⟨trait-name⟩([⟨type-to-sort-list⟩])

⟨type-to-sort-list⟩ → ⟨type-name⟩ **for** ⟨sort-name⟩[,⟨type-name⟩ **for** ⟨sort-name⟩]...

⟨supertypes-clause⟩ → **supertypes** ⟨type-name⟩[,⟨type-name⟩]...

⟨invariant-clause⟩ → **invariant** ⟨predicate⟩

⟨type-body⟩ → ⟨method-specification⟩...

The nonterminals ⟨trait-name⟩ and ⟨sort-name⟩ are just ⟨identifier⟩.

A.2 Method Specifications

⟨method-specification⟩ → ⟨method-header⟩⟨method-body⟩

⟨method-header⟩ → ⟨message-pattern⟩ [**returns** ⟨formal-declaration⟩]

⟨message-pattern⟩ → ⟨unary⟩|⟨binary⟩|⟨keywords⟩

⟨unary⟩ → ⟨identifier⟩

⟨binary⟩ → ⟨binary-selector⟩⟨formal-declaration⟩

⟨keywords⟩ → ⟨keyword⟩⟨formal-declaration⟩[⟨keyword⟩⟨formal-declaration⟩]...

⟨formal-declaration⟩ → ⟨identifier⟩ <: ⟨type-name⟩

⟨method-body⟩ → [⟨pre-condition⟩][⟨modifies-list⟩]⟨post-condition⟩

⟨pre-condition⟩ → **requires** ⟨predicate⟩

⟨modifies-list⟩ → **modifies nothing**
|**modifies** [**at most**] ⟨identifier⟩[,⟨identifier⟩]...

⟨post-condition⟩ → **ensures** ⟨predicate⟩

The lexical conventions for ⟨binary-selector⟩ and ⟨keyword⟩ are the same as in Smalltalk.

A.3 Predicates

⟨predicate⟩ → ⟨boolean⟩|¬⟨predicate⟩|(⟨predicate⟩)
|⟨predicate⟩⟨connectives⟩⟨predicate⟩|⟨quantified⟩|⟨term⟩ = ⟨term⟩
⟨boolean⟩ → `true|false`
⟨quantified⟩ → ⟨quantifier⟩(⟨identifier⟩:⟨sort-name⟩)`[`⟨predicate⟩`]`
⟨quantifier⟩ → ∀|∃
⟨term⟩ → ⟨special⟩|⟨qualified⟩|(⟨term⟩)
|⟨identifier⟩[(⟨term⟩[,⟨term⟩]...)]
|⟨term⟩⟨infix-operator⟩⟨term⟩
| **if** ⟨predicate⟩ **then** ⟨term⟩ **else** ⟨term⟩
⟨special⟩ → ⟨literal⟩|`self`|**fresh**(⟨term⟩[,⟨term⟩]...)
⟨literal⟩ → ⟨number⟩|⟨character⟩|⟨string⟩|⟨symbol⟩
⟨qualified⟩ → $\langle term \rangle_{pre}|\langle term \rangle_{post}|\langle term \rangle_{any}|\langle term \rangle_{obj}$
⟨connectives⟩ → ∧| ∨ | ⇒ | ⇔

The nonterminal ⟨infix-operator⟩ stands for LSL infix trait functions. Larch/Smalltalk literals (⟨literal⟩) are the same as those of Smalltalk.

B. DEFAULT QUALIFICATIONS FOR FORMALS IN ASSERTIONS

Qualifications are often redundant, so Larch/Smalltalk has certain defaults, depending on the context in which an object appears. In the **invariant** clause, an unqualified self is qualified with the value qualifier *any* by default. In the **requires** and **ensures** clauses, self and unqualified formal arguments are qualified with the value qualifier *pre*. In the **ensures** clause, an unqualified output formal parameter is qualified with the value qualifier *post*. In the **modifies** clause and in the Larch/Smalltalk special predicate **fresh**, one always refers to objects. Hence, in these contexts, the object qualifier (*obj*) is the default qualifier.

ACKNOWLEDGMENTS

Thanks to Tim Wahls and the anonymous referees for their helpful comments on drafts of this paper.

REFERENCES

ALENCAR, A. J. AND GOGUEN, J. A. 1991. OOZE: An object oriented Z environment. In *ECOOP'91 Proceedings of the European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 512. Springer-Verlag, New York, 180–199.

AMERICA, P. 1991. Designing an object-oriented programming language with behavioral subtyping. In *Proceedings of Foundations of Object-Oriented Languages*. Lecture Notes in Computer Science, vol. 489. Springer-Verlag, New York, 60–90.

AMERICA, P. 1990. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of OOPSLA ECOOP'90. SIGPLAN Not. 25*, 10 (Oct.), 161–168.

BEAR, S. 1988. Structuring for the VDM specification language. In *Proceedings of the 2nd VDM-Europe Symposium*. Lecture Notes in Computer Science, vol. 328. Springer-Verlag, New York, 2–25.

BRUCE, K. B. AND WEGNER, P. 1990. An algebraic model of subtype and inheritance. In *Advances in Database Programming Languages*. Addison-Wesley, Reading, Mass., 75–96.

CARDELLI, L. AND WEGNER, P. 1985. On Understanding types, data abstraction and polymorphism. *ACM Comput. Surv. 17*, 4 (Dec.), 471–522.

CARRINGTON, D., DUKE, D., DUKE, R., KING, P., ROSE, G., AND SMITH, G. 1989. Object-Z. An object-oriented extension to Z. In *Formal Description Techniques (FORTE'89)*. North-Holland, Amsterdam, 281–296

CHEON, Y. 1991. Larch/Smalltalk: A specification language for Smalltalk Tech. Rep. TR #91-15, Dept. of Computer Science, Iowa State Univ., Ames, Iowa.

CHEON, Y. AND LEAVENS, G. T. 1994. A gentile introduction to Larch/Smalltalk specification browsers. Tech. Rep. 94-01, Dept of Computer Science, Iowa State Univ., Ames, Iowa. Available by anonymous ftp from ftp.cs iastate.edu and by email from almanac@cs.iastate.edu.

CHEON, Y. AND LEAVENS, G. T. 1993. A quick overview of Larch/C++. Tech Rep 93-18, Dept. of Computer Science, Iowa State Univ , Ames, Iowa. Available by anonymous ftp from ftp.cs.iastate edu and by email from almanac@cs.iastate.edu.

CLERICI, S., AND OREJAS, F. GSBL: An algebraic specification language based on inheritance 1988. In *ECOOP'88, European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 322. Springer-Verlag, New York, 78–92

COOK, W. R 1992. Interfaces and specifications for the Smalltalk-80 collection classes. In *Proceedings of OOPSLA '92. SIGPLAN Not. 27*, 10 (Oct.), 1–15.

COOK, W. R 1989. A proposal for making Eiffel type-safe. In *ECOOP'89, European Conference on Object-Oriented Programming*. Cambridge University Press, UK, 57–70.

DAHL, O.-J. 1987 Object-oriented specifications. In *Research Directions in Object-Oriented Programming*. MIT Press, Cambridge, Mass., 561–576.

ERNST, G. W., NAVLAKHA, J. K., AND OGDEN, W. F. 1982. Verification of programs with procedure-type parameters *Acta Informatica 18*, 2, 149–169.

GOGUEN, J. A. AND MESEGUER, J. 1987. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. In *Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., 18–29.

GOLDBERG, A. AND ROBSON, D. 1983. *Smalltalk-80. The Language and Its Implementation*. Addison-Wesley, Reading, Mass.

GUTTAG, J. V. AND HORNING, J. J 1993. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York.

GUTTAG, J. V. AND HORNING, J. J. 1991. Introduction to LCL, a Larch/C interface language. Tech. Rep. 74, Digital Equipment Corporation, Systems Research Center. Palo Alto, Calif.

GUTTAG, J. V., HORNING, J. J., AND WING, J. M. 1985. The Larch family of specification languages. *IEEE Softw. 2*, 4 (Sept).

HAYES, I , ED. 1987. *Specification Case Studies*. International Series in Computer Science Prentice-Hall, Englewood Cliffs, N.J.

HOARE, C. A. R. 1972. Proof of correctness of data representations *Acta Informatica 1*, 4, 271–281.

HOARE, C A. R. 1969. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct) 576–583.

JONES, K. D. 1991. LM3: A Larch interface language for Modula-3, a definition and introduction version 1.0. Tech. Rep 72, Digital Equipment Corporation, Systems Research Center, Palo Alto, Calif.

JUNGCLAUS, R., SAAKE, G., AND SERNADAS, C. 1991. Formal specification of object systems. In *TAPSOFT'91 Proceedings of the International Joint Conference on Theory and Practice of Software Development*. Lecture Notes in Computer Science, vol. 494. Springer-Verlag, New York, 60–82.

LALONDE, W. R. 1989. Designing families of data types using examplars. *ACM Trans Program Lang. Syst. 11*, 2 (Apr.) 212–248.

LALONDE, W. R., THOMAS, D. A., AND PUGH, J. R. 1986. An exemplar based Smalltalk. In *Proceedings of the OOPSLA'86 Conference. SIGPLAN Not. 21*, 11 (Nov) 322–330.

LEAVENS, G. T. 1993. Inheritance of interface specifications (extended abstract). Tech. Rep. 93-23, Dept. of Computer Science, Iowa State Univ. Ames, Iowa. Available by anonymous ftp from ftp.cs.iastate.edu or by email from almanac@cs.iastate.edu.

LEAVENS, G. T. 1991. Modular specification and verification of object-oriented programs. *IEEE Softw. 8*, 4 (July), 72–80.

LEAVENS, G. T. AND CHEON, Y. 1992. Preliminary design of Larch/C++. In *Proceedings of the 1st International Workshop on Larch*. Workshops in Computing Science, Springer-Verlag, New York.

LEAVENS, G. T. AND WEIHL, W. E. 1990. Reasoning about object-oriented programs that use subtypes (extended abstract). In *Proceedings of OOPSLA ECOOP'80. SIGPLAN Not. 25*, 10 (Oct.), 212–223.

LISKOV, B. AND WING, J. M. 1993a. A new definition of the subtype relation. In *ECOOP'93—Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 707. Springer-Verlag, New York, 118–141.

LISKOV, B. AND WING, J. M. 1993b. Specifications and their use in defining subtypes. In *Proceedings of OOPSLA'93 SIGPLAN Not. 28*, 10 (Oct.), 16–28.

MAYER, T. 1988. Specification of object-oriented systems in LOTOS. In *Formal Description Techniques (FORTE'88)*. North-Holland, Amsterdam, 107–119.

MEYER, B. 1988a. Eiffel: A language and environment for software engineering. *J. Syst. Softw. 8*, 3 (June), 199–246.

MEYER, B. 1988b. *Object-oriented Software Construction*. International Series in Computer Science, Prentice-Hall, Englewood Cliffs, N.J.

REYNOLDS, J. C. 1980. Using category theory to design implicit conversions and generic operators. In *Proceedings of a Workshop on Semantics-Directed Compiler Generation*. Lecture Notes in Computer Science, vol. 94. Springer-Verlag, New York, 211–258.

SCHAFFERT, C., COOPER, T., BULLIS, B., KILIAN, M., AND WILPOLT, C. 1986. An introduction to Trellis/Owl. In *Proceedings of OOPSLA '86. SIGPLAN Not. 21*, 11 (Nov.), 9–16.

SCHUMAN, S. AND PITT, D. 1986. Object oriented subsystem specification. In *Proceedings of the IFIP TC2 / WG 2.1 Working Conference on Program Specification and Transformation Program Specification and Transformation*. North Holland, Amsterdam, 313–342.

TAN, Y. M. 1992. Semantic analysis of Larch interface specifications. In *Proceedings of the 1st International Workshop on Larch*. Workshops in Computing, Springer-Verlag, New York.

WILLS, A. 1992. Specification in Fresco. In *Object Orientation in Z*. Workshops in Computing, Springer-Verlag, Cambridge, UK, 127–135.

WING, J. M. 1990. Using Larch to specify Avalon/C++ objects. *IEEE Trans. Softw. Eng. 16*, 9 (Sept.), 1076–1088.

WING, J. M. 1987. Writing Larch interface language specifications. *ACM Trans. Program. Lang. Syst. 9*, 1 (Jan.), 1–24.

WING, J. M. 1983. A two-tiered approach to specifying programs. Tech. Rep. TR-299, MIT, Lab. for Computer Science, Cambridge, Mass.

Received October 1993; revised April 1994; accepted May 1994