A Quick Overview of Larch/C++

Yoonsik Cheon  
Iowa State University

Gary T. Leavens  
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Programming Languages and Compilers Commons

Recommended Citation

http://lib.dr.iastate.edu/cs_techreports/35
A Quick Overview of Larch/C++

Yoonsik Cheon and Gary T. Leavens

TR #93-18a
June, 1993 (Revised June 1994)

Keywords: interface specification, Larch/C++, C++, subtype, specification inheritance, verification.


Submitted for publication.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA
A Quick Overview of Larch/C++

Yoonsik Cheon and Gary T. Leavens*
Department of Computer Science, Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1040 USA
cheon@cs.iastate.edu and leavens@cs.iastate.edu

June 14, 1994

Abstract
This paper gives a quick overview of Larch/C++, an interface specification language for C++. Through examples, we explain declarations, function specifications, class specifications, and template specifications. An extended example is given in the last section. The reader is assumed to have some familiarity with C++. The reader should have some familiarity with the idea of formal specification, but is not required to be familiar with the Larch approach to formal specification.

1 Introduction

Object-oriented programming languages, such as C++ [26], are good for building reusable components. The reuse of program components requires adequate documentation. The specification language Larch/C++ allows interfaces of C++ classes and functions to be documented in a way that is unambiguous and concise. While formal specifications written in Larch/C++ do not replace informal, English documentation, they do provide a precise reference for users of C++ classes and functions. Larch/C++ specifications can be written at a high level, so that the reader does not get bogged down in implementation details. High level specifications also avoid overly constraining implementors.

This paper describes Larch/C++ for potential users. It gives examples to explain how to specify functions, classes, and templates in Larch/C++.

Larch/C++ is an interface specification language. An interface specification language is not tailored to specifying the behavior of an entire program; instead it is tailored to specifying the interface and behavior of a part of a program (a module). For example,

---

*The work of both authors is supported in part by the National Science Foundation under the Grant CCR-9108654.
Larch/C++ specifies C++ classes and functions. The restriction to C++ allows Larch/C++ to have a syntax and semantics that is tailored to C++; for example, the Larch/C++ specification of a C++ function specifies not only the behavior of the function, but exactly how that function is called from C++ code. The details of how to call a C++ function, the name, return type, and argument types, are called the interface of that function. Since Larch/C++ specifies both a C++ interface and behavior, it is an *interface specification language* [29] [27] [16] [10].

The word “Larch” in Larch/C++ refers to the use of the Larch Shared Language [10] to precisely describe behavior. The approach dates back to Hoare’s work [13] on verification of programs that use abstract data types (ADTs). Hoare’s idea was to describe the behavior of an operation of an ADT, not in terms of the bits in the representation of an ADT, but in terms of “abstract values” such as mathematical sets or stacks. This idea lead to the specification languages VDM [14] and Z [12] [25] and the family of Larch interface specification languages [29] [11] [27] [6] [8] [9] [15] [10]. This style of specification is called “model-oriented” in [28], because behavior is described by abstract values taken from some mathematical model.

In Larch/C++ an interface specification consists of two parts: some Larch/C++ specific text, and some text in the Larch Shared Language (LSL) [10]. The LSL text is called a *trait*; it describes the abstract values, and some vocabulary that is used to manipulate those abstract values, that are used in the Larch/C++ part. Each Larch/C++ specification uses a trait in this way. LSL is called a “shared language” because it also plays the same role for other Larch family languages, such as Larch/C [10], Larch/Modula-3 [15][10], and Larch/Smalltalk [6].

For example, Figure 1 shows a simple Larch/C++ specification. The first line is like a C++ #include line; it says that the interface specification in the file IntStack.1cc is to be read for type information. Presumably, IntStack.1cc specifies the class IntStack (see Figure 8). The trait used by this specification is described in the next line, beginning with uses. This specifies that the trait StackTrait from the file StackTrait.lsl (see Figure 7) is to be used to define the abstract values of stacks and trait functions such as size and pop.

The rest of Figure 1 describes the interface and behavior of the C++ function pop_twice.
The interface that this example presents to other C++ code is a function named \texttt{pop\_twice} that has no result (the return type is \texttt{void}), and takes one argument of type \texttt{IntStack\&} (a reference to an \texttt{IntStack} object). The C++ interface is specified with C++ syntax.

The behavior of \texttt{pop\_twice} is specified using a \textit{pre-condition} (starting with \texttt{requires}), a statement of what objects the function is allowed to modify (starting with \texttt{modifies}), and a \textit{post-condition} (starting with \texttt{ensures}). More details on the meaning of these will be given below. For now, suffice it to say that the pre-condition describes what should be true of the abstract value of \texttt{s} before the function is called. The abstract value of the object \texttt{s} in the state before the function is called is denoted \texttt{s\^}. The post-condition describes the relationship between \texttt{s\^} and the abstract value of the object \texttt{s} after the execution of \texttt{pop\_twice} terminates, written \texttt{s'}.

In Eiffel [22, Chapter 7] [23, Chapter 9], one can also use a model-oriented style, but there is no separate mathematical specification of abstract values. That is, there are no traits. Instead, the concrete values of the programming language play the role of abstract values. Instead of using trait functions and LSL terms to express the pre- and post-conditions, in Eiffel one uses Boolean-valued Eiffel expressions.

Distinguishing features of Larch/C++ from other Larch interface specification languages are as follows.

- \textbf{Inheritance of specification.} A derived class (subclass) inherits its base classes’ (superclasses’) specifications. (Inheritance is also present in Larch/Smalltalk [6] and Larch/Modula-3 [15].)
Multiple interfaces for class. A class specification has three different interfaces: one for clients (public), another for derived classes (protected), the third for the implementor and friends (private).

Compatibility with LCL [8], a Larch interface specification language for C. Most parts of LCL specifications are legal Larch/C++ syntax and have the same meaning.

A discussion of our reasons for making certain decisions in the design of Larch/C++ are described elsewhere [18]. In this paper we confine ourselves to informally describing how to write Larch/C++ specifications.

In the following section, Larch/C++ interface modules are explained. In Section 3, declarations are described. In Section 4 and Section 5, function specifications and class specifications are explained. Section 6 describes template specifications in Larch/C++. An extended Larch/C++ example specification is given in Section 7.

2 Larch/C++ Interface Modules

A Larch/C++ specification typically consists of several specification units called interface modules. An interface module is the unit of specification in Larch/C++ and is stored in a separate file. As in LCL [8], some part of an implementation file will be automatically generated from the corresponding interface module. For example, suppose we have an interface module named foo.lcc. A header file named foo.lh will be automatically generated from foo.lcc by planned Larch/C++ tools. This file will have most of the header information that is needed for implementations. The implementor needs to provide foo.h and foo.C; foo.C is a C++ source file and foo.h is its header file. The header file foo.h will provide all the information that is needed to use foo.C, but in general some parts of foo.h cannot be automatically extracted from foo.lcc. Thus the file foo.h will include foo.lh and add any needed parts. Figure 2 shows the relationships among various files mentioned above.

A Larch/C++ specification file, such as foo.lcc, may use several traits in general, not just the trait in foo.lsl.

Figure 3 shows a skeleton of a Larch/C++ interface module. The interface module is named DirectedGraph and is stored in a file named DirectedGraph.lcc; the interface module name is not explicitly specified, but is the same as the file name with its suffix.
.lcc removed. An interface module typically consists of imports clauses, uses clauses, declarations (constants, variables, types, etc.), function specifications, and class specifications. An interface module can import other interface modules. When an interface module is imported, only its public (non-private) definitions are made available to the importing interface module. That is, importing is not simply textual inclusion. The interface module DirectedGraph imports another interface module named Graph, thus all the specifications in Graph.lcc will be visible in DirectedGraph.lcc, except for its private specifications. This is needed, for example, so that the class name Graph can be used in the specification of DirectedGraph. If some other interface module imports the interface module DirectedGraph, then all the specifications in DirectedGraph are exported to that interface module except for the private const declaration MAX_NUM_OF_NODES.

The uses clause in Figure 3 lists the shared components; in the example this is the trait DirectedGraphTrait stored in the file DirectedGraph.lsl. The DirectedGraph interface module specifies an integer constant MAX_NUM_OF_NODES of value 100 and an external function sort. It also specifies a function named foo and a class DirectedGraph, which is a publicly derived class of class Graph.

3 Declarations

A declaration introduces one or more names into a Larch/C++ specification and specifies how those names are to be interpreted. In Larch/C++, one declares constants, variables, types, and functions. The syntax for a declaration in Larch/C++ is the same as that of C++ except that only LSL terms are allowed in the specification of optional initial values. By default, declarations in an interface module are exported. One can write a private
// imports from file Graph.lcc
imports Graph;

// the used trait is in the file DirectedGraphTrait.lsl
uses DirectedGraphTrait;

// declarations
private const int MAX_NUM_OF_NODES = 100; // private to this interface module
extern void sort(float[]);

// function specification
void foo(int& i, int j) {
    modifies i;
    ensures i' = j + 1;
}

// class specification
class DirectedGraph: public Graph {
    // body (omitted)
};

Figure 3: A skeleton of Larch/C++ interface module DirectedGraph.lcc
declaration by preceding it with the keyword `private`. Such declarations must appear in the implementation, and are used to record implementation decisions [18].

### 3.1 Constant Declarations

A Larch/C++ constant declaration is like a `const` variable declaration of C++ with an initializer. The initializer must be an LSL term. Since there are built-in traits for all C++ literals (integers, floating point numbers, characters, and strings), all C++ literals and their operations can be used in the initializer.

```c++
const int MAX_SIZE = 100;
const int MAX_INDEX = MAX_SIZE - 1;
const char str[] = "This is a string";
```

Both `MAX_SIZE` and `MAX_INDEX` are declared as integer constants with values 100 and 99 respectively; if the type specifier is omitted as in `MAX_INDEX`, it is assumed to be `int`. The constant `str` is declared to be a string constant.

In general, a Larch/C++ constant declaration must be implemented by a C++ constant declaration. If the Larch/C++ constant declaration uses an arbitrary LSL term for the initializer, then the C++ initializer will, in general, be a C++ expression. For example, in the Larch/C++ constant declaration

```c++
const Set zoSet = insert(insert(empty,0),1);
```

the identifier `zoSet` is declared to be a constant `Set` with abstract value

```c++
insert(insert(empty,0),1).
```

That is, its abstract value is a mathematical set with two elements 0 and 1. This might be implemented by a C++ constant declaration such as the following.

```c++
const Set zoSet = (Set().add(1)).add(0);
```

A constant declaration can be private; a private constant declaration is not exported (visible) when the interface module in which it is declared is imported to another interface module. For example,
private const int MAX_SIZE = 100;

d Declares a private integer constant MAX_SIZE of value 100.

A constant declaration can be extern. The extern clause says that the actual declaration is in some other interface.

extern const int MAX_NUM_OF_NODES;

Here MAX_NUM_OF_NODES is declared to be an external integer constant.

3.2 Variable Declarations

Syntactically, a Larch/C++ variable declaration is the same as in C++; it consists of type specifiers, declarators, and optional initializers. The optional initializers must be LSL terms. For example:

int i, j = 0;
float x[100];
node *nodeptr;
Set fooSet = insert(empty, i);

A variable can be declared as private, volatile, or extern. By default all variable declarations are visible to the importing interface module when the interface module in which it is declared is imported by another interface module. The keyword private is used to make a variable declaration internal to the interface module in which it is declared. It is precisely equivalent to static which can also be used. The keywords volatile and extern have their usual C++ meaning.

private int i;
extern int j;
volatile int k;

These declarations specify i to be private, j to be extern, and k to be volatile.
3.3 Function Declarations

Like variable declarations, Larch/C++ function declarations are the same as those of C++ function declarations. A function declaration can be private or extern.

```c
int floor(float);
extern void sort(int[], int);
private int max(int, int);
```

Here `floor` is declared to be a function that takes a `float` and returns an `int`, and `sort` is declared to be an external function that takes two arguments, an `int` array and an `int` denoting the size of the array, and returns nothing. The private function `max` takes two integers and returns an integer. A private function declaration records a design decision, but is of no concern to clients.

3.4 Type Declarations

A new type can be introduced into a Larch/C++ specification by `typedef`, `enum`, `struct`, `union`, or `class`. The syntax for type declarations with `typedef` and `enum` is the same as that of C++. Class specifications (`class`, `struct`, and `union`) are explained in Section 5.

```c
typedef char /*string*/;
enum color {red, yellow, green, blue};
```

In the above example, the type `string` is a synonym for `char *` (character pointer). Each enumeration defines a new integral type that is different from all other integral types. Thus type `color` is a new integral type with four enumerators: `red`, `yellow`, `green`, and `blue`.

4 Function Specifications

A function is specified with Hoare-style pre- and post-conditions. The header of a function specification is the same as that of a C++ function definition; that is, one specifies the optional return type, name of the function, and formal arguments if any. The body describes the effect of a function invocation by a pair of predicates following the keywords `requires` and `ensures`. The predicate after `requires` is a pre-condition that must be satisfied to
typedef bool int;
bool sorted_interval(float x[], int l, int u) {
   requires 0 <= l /\ l <= u /\ u <= maxIndex(x);
   ensures result =
   int(\forall i,j:int
       ((l <= i /\ i <= j /\ j <= u) => (x[i] <= x[j])));
}

Figure 4: Specification of the function sorted_interval.

invoke the specified function. The predicate after ensures is a post-condition that the
specified function establishes upon termination of the function invocation. The requires
clause is optional; an omitted requires clause is equal to requires true.

The function sorted_interval, specified in Figure 4, checks if a given interval of an
array is sorted. The pre-condition says that l and u together denote a legal slice of the
array x. The notation /\ means conjunction. The trait function maxIndex denotes the
upper bound of an array [10]. It is part of a trait for arrays that is built-in to Larch/C++. (Traits for the built-in types of C++ are automatically used by a Larch/C++ specification,
so no uses clause is needed in this case.) The post-condition states that result is an
integer encoding of true if all the elements in the interval are in order; otherwise it is 0.
The trait function int\(^1\) in the post-condition comes from the Larch/C++ built-in trait int,
and converts a boolean value to an int. The Larch/C++ built-in traits (one for each C++
built-in type and a few others) are implicitly available to all Larch/C++ specifications.
The syntactic form /\forall is the universal quantifier, and => means implication. The
Larch/C++ keyword result denotes what is returned by the specified function. The type
of result is the return type of the specified function; e.g., result is of type bool.

In the specification of sorted_interval, the arguments names l, u are passed by value,
and so are not considered objects. (Although l and u have locations in a C++ program,
changes to these locations are not visible to the caller of the procedure, and so Larch/C++
does not model them as objects. This follows LCI [10].) Similarly the universally quantified
variables i and j are also names of abstract values, not names of objects. However, since

\(^1\)The identifier int is used in four different ways: (1) type name, (2) trait function, (3) sort name, and
(4) trait name. Since these name spaces are distinct in Larch/C++, the meaning is clear from context.
x is an array of floats, \( x[i] \) and \( x[j] \) are objects (i.e., locations), and hence to get their abstract values before the \texttt{sorted_interval} is called, one must write \( x[i] \uparrow \) and \( x[j] \uparrow \).

Both the pre- and post-conditions are predicates in the first-order predicate calculus specified in terms of arguments, \texttt{result}, and trait functions of the used traits, and some built-in traits. The built-in traits allow the use of C++ literals, and boolean and arithmetic operators.

The semantics of function specifications is that the function's state transformation must satisfy the formula: \texttt{pre-cond} \( \Rightarrow \) \texttt{post-cond}. That is, if the \texttt{pre-cond} is satisfied in the pre-state (the state just before function invocation), then the function evaluation terminates and the \texttt{post-cond} is satisfied in the post-state (the state just after function invocation). If \texttt{pre-cond} is not satisfied in the pre-state, nothing is guaranteed in the post-state.

For functions that change the values of objects (e.g., by assignment), the body of their function specification must include a \texttt{modifies} clause. The \texttt{modifies} clause asserts that only those objects listed may change their abstract values as the result of function invocation. This is a strong indirect assertion that no other objects, except for those listed in the clause, are allowed to change their values as the result of function evaluation. An omitted \texttt{modifies} clause is equal to \texttt{modifies nothing}, meaning no arguments can be changed, nor can any other global object.

As an example, the function \texttt{swap} (see below) can mutate both of its two arguments \( i \) and \( j \), but nothing else. For example, \texttt{swap} cannot mutate any global variables.

```c++
void swap(int& i, int& j) {
    modifies i, j;
    ensures (i' = j\uparrow) \&\& (j' = i\uparrow);
}
```
In a function that mutates an object or a variable, it is necessary to distinguish two different values of the same object; the value in the pre-state and the value in the post-state. The value of an object or variable in the pre-state, called its pre-value, is denoted by a hat-ed (\(\hat{\cdot}\)) identifier while the post-state value, called post-value, is represented by a primed (\(\prime\)) identifier. For example, in the post-condition of the function swap, \(\hat{i}\) denotes the pre-value of \(i\), \(i'\) stands for the post-value of \(i\). The post-condition of swap says that the post-value of \(i\) is equal to the pre-value of \(j\) and post-value of \(j\) is equal to the pre-value of \(i\); that is, the values of \(i\) and \(j\) are swopped. The notation \(i\) in the modifies clause means the \(i\) object itself, i.e., the memory location. The name \(i\) is an object because \(i\) is declared to be a reference parameter. Non-reference parameters are considered to be values, since C++ passes them by value, and such names do not change their meaning in the pre- and post-states of a function call (as seen by the caller).

Default values of formal arguments can be given in function specifications. The syntax is the same as that of C++ except that the expressions giving the default values must be ISL terms. As with constant declarations, such an ISL term must, in general, be implemented by a C++ expression. However, in the following example, the ISL term is also a C++ literal.

```c++
float interest(float x, float rate = 0.05)
{
    requires 0.0 <= rate /\ rate <= 1.0;
    ensures result = x * rate;
}
```

The function interest takes two float values denoted by \(x\) and \(rate\) respectively, and computes interest based on \(rate\). If no value is supplied for \(rate\) on function invocation, 0.05 is used by default.

A global variable can be referenced in a C++ function. In a Larch/C++ function specification, all the global variables referenced in a function must be explicitly listed. Listing such external variables is an aid to program verification and understanding.

In Figure 5, the global variable \(db\) is referenced in the function search, thus it is explicitly declared to be an extern variable. After their declaration, global variables can be freely used in requires, modifies, and ensures clauses. The predicate legalIndex is
const int SIZE = 100;
float db[SIZE];

int search(float x)
{
    extern float db[];

    ensures if \exists i:int (legalIndex(db,i) \&\& db[i]^- = x)
    then (legalIndex(db,result) \&\& db[result]^- = x)
    else result = -1;
}

Figure 5: Larch/C++ specification of a function search.

a Larch/C++ built-in trait function which is true if the second argument is a legal index to
the first argument, array; i.e., the axiom for legalIndex is:

legalIndex(db,i) == (0 <= i \&\& i <= maxIndex(db)).

5 Class Specifications

In C++, a class introduces a new abstract data type with encapsulated data and operations
(member functions). A Larch/C++ class specification is similar to C++ class definition.
Data members are declared with the syntax shown in Section 3. Similarly, member functions
are specified exactly the same way as stand-alone functions (see Section 4) except that one
can use the variable this and the pseudo-variable self in predicates. The Larch/C++
reserved word this means the same thing as the C++ reserved word this, a pointer to the
object of specified class, and self is a shorthand for *(this\any). The suffix \any is like
' or ^, and extracts the value of this in some visible state; since in a member function
of a class named X, this is implicitly declared as X *const this [26, Section r.9.3.1], it
is a constant, and so its value is the same in all states. Both data members and member
functions can be public, protected, or private.

Figure 6 shows a Larch/C++ class specification named IntStack. The uses clause
tells Larch/C++ that the specification is expressed with the vocabulary of the ISL trait
StackTrait (see Figure 7). The trait StackTrait defines a mathematical model of stacks.
All the terms in pre- and post-conditions of function specifications come from this trait.
class IntStack {

    uses StackTrait(IntStack for S, int for E);

public:
    IntStack() { // constructor
        modifies self;
        ensures self' = empty;
    }

    ~IntStack() { // destructor
        modifies self;
        ensures trashed(self);
    }

    IntStack& push(int i) {
        modifies self;
        ensures self' = push(self~,i) \ result = self;
    }

    IntStack& pop() {
        requires "isEmpty(self~);"
        modifies self;
        ensures self' = pop(self~) \ result = self;
    }

    int top() const {
        requires "isEmpty(self~);"
        ensures result = top(self~);
    }

    int isEmpty() const {
        ensures result = int(isEmpty(self~));
    }
};

Figure 6: A class specification IntStack
StackTrait(S,E): trait
includes int % describes values of C++ integers
introduces
    empty: -> S
    push: S,E -> S
    pop: S -> S
    top: S -> E
    size: S -> int
    isEmpty: S -> Bool
asserts
S generated by empty, push
\forall s, s1: S, e: E
    top(push(s,e)) == e;
    isEmpty(empty);
    ~isEmpty(push(s,e));
    pop(push(s,e)) == s;
    size(empty) == 0;
    size(push(s,e)) == (1 + size(s))

Figure 7: A trait StackTrait

The type-to-sort mapping, which is given between the parentheses following the names of the used trait, says that the abstract values of C++ IntStack objects in a program are specified to be those of the LSL sort S in StackTrait. (In LSL, a sort is the “type” of an LSL term; we use the word “type” only to refer to C++ types, either in C++ programs or in Larch/C++ specifications.) The type-to-sort mapping makes the connection between the C++ world and the LSL (mathematical) world.

Some technical remarks are in order here. First, the type-to-sort mapping actually maps type names to sets of sort names. For example, the C++ type int is already mapped to the LSL sort int, and so the uses clause in IntStack maps the type int to the set \{int, E\}. This means that the sorts int and E are identified. In other words, the sort E in StackTrait should be interpreted as the sort int. Second, the abstract values specified by a trait such as StackTrait are technically equivalence classes of LSL terms, where the notion of equivalence is specified by the trait. For example, the abstract values of IntStack objects are equivalence classes of the LSL terms of sort S from StackTrait. Example abstract values are the equivalence classes of the terms: empty, push(empty,0), push(push(empty,0),1),
and so on.

Figure 6 specifies a constructor, a destructor, and four public member functions: `push`, `pop`, `top`, and `isEmpty`. As mentioned before, "self" is short for "*(this\any)". The destructor uses the Larch/C++ reserved word `trashed` to state that the object `self` is no longer available. Note that identifiers like `push`, `pop`, `top`, and `isEmpty` are used as both C++ function names and as trait functions. However, the context in which they appear tells whether they are C++ functions or trait functions; that is, those in the pre- and post-conditions are trait functions. The meanings of the trait functions are precisely defined in the used trait `StackTrait`. The symbol `~` is used for logical negation (¬).

All the declarations discussed in Section 3 can appear in class specifications. Data members can be `static` with its usual C++ semantics. Member functions can be `virtual`, `static`, or `inline`; these all have their C++ meanings. As one might guess, a class can be specified to have `friend` functions or classes.

It is a good practice to specify a class by itself in an interface module, because a class specification is convenient way to encapsulate a concept, and because subtyping and inheritance of specifications is a good way of organizing Larch/C++ specifications.

## 5.1 Inheritance of Specifications

All Larch specification languages can reuse LSL traits. Unlike most other Larch specification languages, however, Larch/C++ interface specifications can be reused by inheritance. A derived class (i.e., a subclass) inherits its base classes’ specifications; that is, data member declarations and member function specifications (and invariants) of the base classes are inherited by the derived class. Specification inheritance provides a simple, flexible mechanism for specifying a class by adding additional properties without respecifying existing classes. As in C++, a class can be specified to be a `public`, `protected`, or `private` derived class of its base classes. The semantics is similar to that of C++; for example, if a class `S` is a `private` derived class of class `T`, then no member functions inherited from `T` are visible to a client of class `S` even though they are specified to be `public` in class `T`.

The Larch/C++ specification of a derived class requires that the C++ implementation must be derived in the same way. That is, if a class `S` is specified to be a `public` derived class of a class `T`, the class that implements `S` should be a `public` derived class of the class that
implements T. Similarly, the specification of a private base class records a design decision that must be respected in implementations. However, an implementation may have more private base classes than specified.

As in C++, the specification of a public base class makes pointers and references to the derived class subtypes of pointers and references to the base class. To make reasoning about a program that uses message passing and subtyping tractable, one should ensure that the member functions of the base class have an appropriate specification in the derived class [24] [1] [19] [3] [17]. This is almost automatic if one inherits the specifications, except that one has to ensure that the trait functions used in the base classes' specifications have the appropriate meaning [18]. See the references for more details.

Larch/C++ allows multiple inheritance. A class can be specified to have more than one direct base class. The ambiguity resolution rules for multiple occurrence of a base class are the same as those of C++.

An example of derived classes and specification inheritance can be found in Section 7.

5.2 Abstract Base Classes

There are many cases in which classes are used to represent abstract concepts for which objects may not exist. They are intended to be base classes of concrete derived classes. In C++, a class with a pure virtual function (a virtual function with an initializer = 0) is defined to be an abstract class, and no object of this abstract class can be created. In Larch/C++, we specify an abstract class by writing the keyword abstract class instead of just class.

The syntax for specifying data members and member functions is the same as that of concrete class specifications. However, no specifications of constructors or destructors are allowed, because no objects of abstract classes can be created.

For an abstract class, the usual rule that all specified member functions must be implemented is relaxed in the sense that a pure virtual function can be a legal implementation of a member function specification of the abstract class. An example of an abstract class specification, Graph, is given in Figure 13.
template <class Elem>
class Stack {

    uses StackTrait(Stack<Elem> for S, Elem for E);

public:
    Stack() {  // constructor
        modifies self;
        ensures self' = empty;
    }

    ~Stack() {  // destructor
        modifies self;
        ensures trashed(self);
    }

    Stack<Elem>& push(Elem e) {
        modifies self;
        ensures self' = push(self^,e) \ result = self;
    }

    Stack<Elem>& pop() {
        requires ~isEmpty(self);
        modifies self;
        ensures self' = pop(self^) \ result = self;
    }

    Elem top() const {
        requires ~isEmpty(self);
        ensures result = top(self);
    }

    int isEmpty() const {
        ensures result = int(isEmpty(self));
    }
};

Figure 8: A template class specification Stack
6 Templates

A template specifies a family of classes or functions. Template classes are mostly used to specify general container types such as Set, Stack, and List where the specific elements type is left unspecified as a parameter. For example, instead of specifying classes IntStack, CharStack, StringStack and so on, one can specify a template Stack once and for all. Then, the template can be instantiated for arbitrary types to produce new class specifications such as Stack<int>, Stack<char>, Stack<string> and so on.

A template version of the function search (see Figure 5) may be specified as follows.

```cpp
template<class Elem>
int search(Elem arr[], Elem x) {
    ensures if \exists i:int (legalIndex(arr,i) \ arr[i] = x)
        then (legalIndex(arr,result) \ arr[result] = x)
    else result = -1;
}
```
As in C++, a template specification is preceded by the keyword `template` and template arguments enclosed in a pair of angle braces (`<` and `>`). The arguments are type variables that are left unspecified, and later instantiated to actual types. Everything else is the same as a normal function specification except that the type variables can be freely used in the specifications; they are treated as type names. For example, the type variable `Elem` was used to specify the argument types.

Figure 8 shows a template version of the `IntStack` specification. Now the member function `push` takes a value of type `Elem` and returns a reference to a `Stack<Elem>` object. If a variable `s` is declared to be of type `Stack<int>`, then `s.push` will take an integer as its argument and return a reference to an integer stack (`Stack<int>&`) as the result. Instantiations (`Stack<int>, Stack<char>, etc`) are regarded as class names; thus they can be used any place where class names are required. For example, they can be used to declare variables, to specify argument and return types of function specifications, or to specify base classes in class specifications.

Implementations of template function/class specifications must be C++ template functions/classes.

7 An Extended Example

In this section, we specify several Larch/C++ interface modules for two types of graphs: undirected graphs and directed graphs.

Mathematically, a graph $G$ is an ordered triple $(N(G), E(G), \psi_G)$ consisting of a nonempty set $N(G)$ of nodes (vertices), a set $E(G)$ of edges, which is disjoint from $N(G)$, and an incidence function $\psi_G$ that associates with each edge of $G$ a pair of (not necessarily distinct) nodes of $G$. If the edges are ordered, the graph is directed; otherwise it is undirected. For directed graphs, we use the term arcs instead of edges. To make things simple in our traits for graphs, the incidence function will be ignored and an edge is modeled by a tuple of nodes. The first element of the tuple is called the head and the second element is called the tail. Thus, in our model there can be only one edge (or arc) from a node to another node. In our traits, we also model an empty graph, a graph with no nodes.

We will specify two classes, `DirectedGraph` and `UndirectedGraph`, which describe directed graphs and undirected graphs respectively. To take benefit of specification inher-
GraphTrait(N,E,G): trait
includes Set(N,SN), Set(E,SE) % from LSL handbook
G tuple of nodes: SN, edges: SE
E tuple of head: N, tail: N
introduces
    includesNode, isolatedNode: G, N -> Bool
asserts
\forall g, g1:G, sn: SN, se: SE, n,m,m1: N
    includesNode(g,n) == n \in g.nodes;
    isolatedNode([sn,\{\}],n);
    isolatedNode([sn,insert([m,m1],se)],n) == ~(n = m \lor n = m1)
    /\ isolatedNode([sn,se],n)

Figure 9: The trait GraphTrait

itance, we abstract out all those features that are common to both undirected graphs
and directed graphs into an abstract class Graph. Both the class DirectedGraph and
UndirectedGraph are defined to be directly derived from the base class Graph, thus inher-iting all the properties specified in the class Graph.

The trait used by the class Graph is shown Figure 9. A graph is modeled by a tuple
of nodes and edges, where nodes is of sort SN (set of N) and edges is of sort SE (set of
E). 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 LSL shorthand notation for introducing
fixed-length tuples [10]. For example, "G tuple of nodes: SN, edges: SE" is equivalent
to including a trait with body shown in Figure 10.

All the information specified in the trait GraphTrait is only about nodes, since it is
not yet known whether edges are directed or undirected. The trait GraphTrait defines two
trait functions: includesNode and isolatedNode. The trait function includesNode tells
whether a node is in a graph, while isolatedNode tests if a node is isolated from others. The
trait function \in in the axiom for includesNode represents the set membership operation
(\in), which comes from the included trait Set. The trait Set defines a mathematical model
for sets and with typical set operations; \cup (\cup), \setminus (\setminus), - (set-difference), etc. Some of
these set operations are used in the specifications of classes Graph, DirectedGraph, and
UndirectedGraph. The trait Set is found in the Larch Shared Language Handbook [10,
introduces

\[
\begin{align*}
\text{[\_,\_\_]} & : \text{SN, SE} \rightarrow G \\
\text{\_\_.nodes} & : G \rightarrow \text{SN} \\
\text{\_\_.edges} & : G \rightarrow \text{SE} \\
\text{set\_nodes} & : G, \text{SN} \rightarrow G \\
\text{set\_edges} & : G, \text{SE} \rightarrow G
\end{align*}
\]

asserts

\[
\begin{align*}
G & \text{ generated by [\_,\_\_] } \\
G & \text{ partitioned by \_\_.nodes, \_\_.edges} \\
\forall g : G, n, n\_1 : \text{SN}, e, e\_1 : \text{SE} \\
[n, e].\text{nodes} & = n; \\
[n, e].\text{edges} & = e; \\
\text{set\_nodes}[n, e, n\_1] & = [n\_1, e]; \\
\text{set\_edges}[n, e, e\_1] & = [n, e\_1]
\end{align*}
\]

Figure 10: In LSL, writing a tuple definition of the form “G tuple of nodes: SN, edges: SE” is equivalent to including a trait whose body is shown above.

Appendix A]. A node is isolated if the graph has no edge at all or there is no edge between the node itself and the rest of nodes in the graph. The second and the third axioms state this.

The trait `UndirectedGraphTrait` shown in Figure 11 defines a formal model for the class `UndirectedGraph`. The assumption of `GraphTrait` says that `UndirectedGraphTrait` will be used in an interface specification where `GraphTrait` will also be used. This assumption is needed because the sort `G` of `GraphTrait` is used in defining the trait function `toG`, which coerces abstract values of undirected graphs to the abstract values of graphs. The trait `UndirectedGraphTrait` also includes `GraphTrait` in order to “inherit” its definitions; when it does this it renames the sort `G` to `UDG`, so that there are now sorts for graphs and undirected graphs. (Note that the sorts of nodes, edges, and sets of edges are the same.) In addition to properties stated in the included trait `GraphTrait`, it defines a trait function `includesEdge`. An edge `e` is included in an undirected graph `g` if the edge set of `g` (`g.edge`) includes `e` or `[e.tail, e.head]`. This is because the edge `e` has no direction associated with it. The LSL notation `[x, y]` denotes a tuple with the first element `x` and the second element `y`. The trait also has a significant definition of the trait function `\text{eq\_as\_UDG}`, which defines an appropriate notion of equality for terms of sort `UDG`. The definition says that two terms
UndirectedGraphTrait(N,E,UDG): trait
assumes GraphTrait
includes GraphTrait(N,E,UDG)
introduces
toG: UDG -> G
includesEdge: UDG, E -> Bool
__ \eq_as_UDG __: UDG, UDG -> Bool
equalEdges, subsetEdges: UDG, UDG -> Bool

asserts
\forall g,g1:UDG, e: E, n,m: N, sn: SN, se: SE
toG(g) == [g.nodes, g.edges];
includesEdge(g,e) == e \in g.edges
\ /
\ /
(g \eq_as_UDG g1) == (g.nodes = g1.nodes \ / equalEdges(g,g1));
equalEdges(g,g1) == subsetEdges(g,g1) \ / subsetEdges(g1,g);
subsetEdges([sn,{}],g1);
subsetEdges([sn,insert(e,se)],g1) == includesEdge(g1,e)

Figure 11: A trait UndirectedGraphTrait

of sort UDG are considered equal (as UDGs) when they have the same set of nodes and their sets of edges are equal, ignoring direction. The use of = between undirected graphs in a Larch/C++ specification is interpreted by the trait function \eq_as_UDG, so that the abstract values of undirected graphs are properly compared.

A mathematical model for the class DirectedGraph, the trait DirectedGraphTrait, shown in Figure 12 is similar to the trait UndirectedGraphTrait, except that now each arc has a direction attached to it, thus the first axiom in Figure 12. Note that the sort E is renamed to A and the trait function .edges is renamed to .arcs. The trait function toG is a coercion function that maps abstract values of directed graphs to graph abstract values. It is used in the interface specification of DirectedGraph to say how directed graphs simulate graphs.

We now turn to specifying the interfaces of the various kinds of graphs. As graphs are useful with a variety of different nodes, the three classes are specified as templates.

Figure 13 shows the interface module that specifies Graph. The specification of Set (not shown here) is imported, since the nodes operation returns a reference to a set of nodes. The class Graph is specified both as a template and as an abstract class. As an abstract
DirectedGraphTrait(N,A,DG): trait assumes GraphTrait
includes GraphTrait(A for E, .arcs for .edges, DG for G)
introduces
  includesArc: DG, A -> Bool
  toG: DG -> G
asserts
  \forall g:DG, a: A
    includesArc(g,a) == a \in g.arcs;
    toG(g) == [g.nodes, g.arcs]

Figure 12: A trait DirectedGraphTrait

class, Graph has no specifications for constructors or destructors. The type variable Node represents the type of nodes. The specification has five public member functions: addNode, removeNode, chooseNode, nodes, and numOfNodes. Terms in the pre- and post-conditions of these function specifications come from the trait GraphTrait. All member functions are specified to be virtual, thus they must be implemented by C++ virtual member functions. The member functions chooseNode, nodes, and numOfNodes are defined to be const functions. As in C++, a const member function is not allowed to change the representation of the object *this.

Given a node, not already included in a graph, the member function addNode adds the node to the graph. The post-condition says that self' (the post-value of self) is equal (=) to self^ (pre-value of self) with its nodes replaced by the union of self^ .nodes (nodes of self in the pre-state) and the node to be added. The trait function \cup stands for set union (\cup) and comes from the trait Set, which is included by the used trait GraphTrait. The function returns a reference to an object of type Graph<node>. The member function removeNode deletes an existing node from the graph. The pre-condition says it can be invoked only with a node with no edges associated with it; i.e., the argument node should be isolated. The symbol - in the post-condition represents set difference and is defined in the trait Set. The member function chooseNode is interesting in that its post-condition is under-specified. All it says is that the return value is a node of self^ . It does not say which one should be returned if there is more than one node. The implementor has freedom to choose an appropriate algorithm (which might be either non-deterministic or
imports Set; // interface module specifying template class Set

template <class Node>
abstract class Graph {

uses Graph(Graph<Node> for G, Set<Node> for SN, Node for N);

public:
  virtual Graph<Node> & addNode(Node n) {
    modifies self;
    ensures self' = set_nodes(self^n, self^n.nodes \ U \{n})
      \ result = self;
  }

  virtual Graph<Node> & removeNode(Node n) {
    requires includesNode(self^n,n) \ isolatedNode(self^n,n);
    modifies self;
    ensures self' = set_nodes(self^n, self^n.nodes - \{n})
      \ result = self;
  }

  virtual Node chooseNode() const {
    requires ~isEmpty(self^n.nodes);
    ensures includesNode(self^n, result);
  }

  virtual Set<Node> & nodes() const {
    ensures fresh(result) \ result' = self^n.nodes;
  }

  virtual int numOfNodes() const {
    ensures result = size(self^n.nodes);
  }
};

Figure 13: An interface module Graph.lcc
The function `nodes` returns a reference to a new set containing all the nodes of `self`; `fresh` is a Larch/C++ reserved word asserting that the argument is a newly created object. The function `size` returns the number of nodes in the graph. Note that no member function is concerned with edges because it is not yet known whether the edges have directions associated with them or not. These are properties to be specified by concrete derived classes.

Figure 14 shows an interface module `DirectedGraph.lcc`. The class `DirectedGraph` is defined to be a template with a type variable `Node`. The template class `DirectedGraph` is specified to be a direct `public` derived class of the template `Graph`. For each type `T`, `DirectedGraph<T>` is a derived class of `Graph<T>`. For example, `DirectedGraph<int>` is a derived class of `Graph<int>`, and `DirectedGraph<char>` is a derived class of `Graph<char>`. However, `DirectedGraph<int>` is not a derived class of `Graph<char>` or vice-versa. Since the class `DirectedGraph` is a `public` derived class, all the `public` member functions of the class `Graph` are visible to the clients of the class `DirectedGraph`.

The `simulates` clause says that each directed graph abstract value, `d`, simulates the graph abstract value, `toG(d)`. That is, the trait function `toG` can be used as a coercion function from directed graphs to graphs. Having such coercion functions is useful in dealing with subtyping [5] [1] [2] [4] [20] [21]. In Larch/C++, one can specify a trait function that acts as a coercion for each base type. This coercion function helps to define the semantics of inherited Larch/C++ specifications.

The `invariant` clause introduces a predicate that must be preserved by all objects of the specified class. That is, a class invariant is an assertion `p` that holds in the initial state (just after creation of an object) and that is left invariant by each member function `f` of the specified class. If `p` hold in the pre-state and the function `f` transforms the pre-state to a post-state, then `p` will hold in that post-state. As a result, the invariant should hold in all visible states that can be reached from the initial state by means of message passing. As in the pre- and post-conditions, `self` in the `invariant` clause denotes the receiver, an object of the specified class, and `self\any` denotes the value of `self` in any visible state. We use `self` so that the invariant can be thought of as being implicitly conjoined to the pre- and post-conditions of all member function specifications. If no invariant is specified, `true` is assumed by default. The invariant clause is used to restrict the domain of abstract values.
imports Graph;

template<class Node>
class DirectedGraph : public Graph<Node> {
    uses DirectedGraph(DirectedGraph<Node> for DG, Set<Node> for SN, Node for N);
    simulates Graph<Node> by toG;
    invariant \forall n:N, m:N (includesArc(self\any,[n,m]) =>
        includesNode(self\any,n) \& includesNode(self\any,m));

public:
    DirectedGraph() { // constructor
        modifies self;
        ensures self' = [{},{}]; // graph with no nodes & no arcs
    }
    virtual ~DirectedGraph() { // destructor
        modifies self;
        ensures trashed(self);
    }
    virtual DirectedGraph<Node>& addArc(Node n, Node m) {
        requires includesNode(self\any,n) \& includesNode(self\any,m);
        modifies self;
        ensures self' = set_arcs(self\any, self\any.arcs \cup \{[n,m]\}) \& result = self;
    }
    virtual DirectedGraph<Node>& removeArc(Node n, Node m) {
        requires includesArc(self\any,[n1,n2]);
        modifies self;
        ensures self' = set_arcs(self\any, self\any.arcs - \{[n,m]\}) \& result = self;
    }
    virtual Set<Node>& adjacentNodesFrom(Node n) const {
        requires includesNode(self\any,n);
        ensures fresh(result) \& \forall m:N (m \in result' \iff \{n,m\} \in self\any.arcs);
    }
    virtual Set<Node>& adjacentNodesTo(Node n) const {
        requires includesNode(self\any,n);
        ensures fresh(result) \& \forall m:N (m \in result' \iff \{m,n\} \in self\any.arcs);
    }
    virtual Set<Node>& adjacentNodes(Node n) const {
        requires includesNode(self\any,n);
        ensures fresh(result) \& \forall m:N (m \in result' 
            \iff \{n,m\} \in self\any.arcs \& \{m,n\} \in self\any.arcs);
    }
};

Figure 14: An interface module DirectedGraph.lcc
of a class to a subset of the values defined by the used trait. The invariant of the class `DirectedGraph` says that if `[n,m]` is an arc of a directed graph then both `n` and `m` must be nodes of the graph. That is, the abstract values of class `DirectedGraph` is those terms of sort `DG` in the trait `DirectedGraph` (see Figure 12) that satisfy the invariant. For example, `[][[]]` is one possible abstract value, a directed graph with no node and no arc. However, `[][[],{[n,m]}]` cannot be an abstract value of the class `DirectedGraph` even though it is a term of sort `DG`; it does not satisfy the invariant.

The class `DirectedGraph` specifies both a constructor and a destructor. The constructor returns an empty directed graph `[][[]]`, a graph with no nodes and no arcs. The destructor simply asserts that the receiver is trashed; that is, it is not available anymore. The class `DirectedGraph` inherits all the member functions of the class `Graph`. In addition to these inherited member functions, the class specifies five new public member functions: `addArc`, `removeArc`, `adjacentNodesFrom`, `adjacentNodesTo`, and `adjacentNodes`. All are virtual functions. The member function `addArc` adds a new arc, denoted by a pair of nodes, while `removeArc` deletes an existing arc from the graph. Note that the pre-condition of `addArc` requires both the head and tail of the arc to be added to be nodes of the graph `self`. Since `addArc` is the only member function that adds arcs, every object of the class `DirectedGraph` satisfies the class’s invariant. The member function named `adjacentNodesFrom` returns a reference to a new set containing all the nodes which are adjacent from a given node, while `adjacentNodesTo` returns a reference to a new set with all the nodes adjacent to a given node. And the member function `adjacentNodes` returns a set of all the nodes adjacent to and from a given node.

The class `UndirectedGraph`, another directly derived class of the class `Graph`, is shown in Figure 15. It is also a public derived class, thus all the member functions of the class `Graph` are inherited. For each type `T`, `UndirectedGraph<T>` is a public directly derived class of the class `Graph<T>`. The class invariant is similar to that of the class `DirectedGraph`, thus restricting the domain of the abstract values of the class `UndirectedGraph` to a subset of the values of sort `UDG` in the used trait `UndirectedGraphTrait`. The constructor returns an empty undirected graph `[][[]]` and the destructor simply trashes the receiver. The public virtual member function `addEdge` inserts a new edge to the receiver and the function `removeEdge` deletes an existing edge from the receiver. The post-condition of
imports Graph;

template<class Node>
class UndirectedGraph: public Graph<Node> {

  uses UndirectedGraph(UndirectedGraph<Node> for UDG, Set<Node> for S, Node for N);
  simulates Graph<Node> by toG;
  invariant \forall n:N, m: N (includesEdge(self\any,[n,m]) =>
    includesNode(self\any,n) \& includesNode(self\any,m));

public:
  UndirectedGraph() { // constructor
    modifies self;
    ensures self' = [{},{}]; // graph with no nodes & no edges
  }

  virtual ~UndirectedGraph() { // destructor
    modifies self;
    ensures trashed(self);
  }

  virtual UndirectedGraph<Node>& addEdge(Node n, Node m) { // constructor
    requires includesNode(self\^,n) \& includesNode(self\^,m);
    modifies self;
    ensures self' = set_edges(self\^, self\^ .edges \U {{n,m}})
    \& result = self;
  }

  virtual UndirectedGraph<Node>& removeEdge(Node n, Node m) { // destructor
    requires includesEdge(self\^,[n,m]);
    modifies self;
    ensures self' = set_edges(self\^, self\^ .edges - {{n,m}} \U {{m,n}}))
    \& result = self;
  }

  virtual Set<Node>& adjacentNodes(Node n) const { // interface
    requires includesNode(self\^,n);
    ensures fresh(result) \& \forall m:N (m \in result' => ([n,m] \in self\^ .edges) \& ([m,n] \in self\^ .edges));
  }
};

Figure 15: Interface module UndirectedGraph.lcc
removeEdge states that it deletes both the edges \([n, m]\) and \([m, n]\). Because there is no direction associated with an edge, both \([n, m]\) and \([m, n]\) denote the same undirected edge, an edge between nodes \(n\) and \(m\). Both member functions return references to the object self. The const member function adjacentNodes returns a reference to a new set containing all the adjacent nodes of a given node.

8 Related Work

Larch/C++ solves the following problems with Eiffel's style of specification [22, Chapter 7] [23, Chapter 9]. In Eiffel,

1. There is no way to express universally or existentially quantified assertions.

2. One is sometimes forced to export more operations than one would like in order to specify some types. For example, to specify a statistical database with instance operations insert, mean, and variance, one would also need to export operations that enumerate the elements to state the post-condition of insert [23, section 9.8]. However, a designer may wish to hide such operations for other reasons.

3. The meaning of an assertion is unclear if the operations involved fail to terminate or use non-portable parts of the language.

Although Larch/C++ does not suffer from the above problems, it may be more difficult for the average programmer to use and understand than Eiffel's specifications. Only more experience will tell.

A refinement of the Eiffel specification language is found in the specification language Annotated C++ (called A+++) [7]. Assertions in A++ may use universal and existential quantification, and hence are not generally executable, although they are still expressed using the expressions of C++. A difference from Eiffel is that the assertions must be expressed in a pure subset of C++, which makes them more amenable to formal manipulation and solves the first problem above. Furthermore, besides pre- and post-conditions for functions, in A++ one can give pre- and post-conditions for blocks of C++ code, which allows one to specify that

\[ s \rightarrow \text{push}(x); \ s \rightarrow \text{pop}(); \]
does not change s. While A++ is more expressive than Eiffel's specification sublanguage, it still suffers from all but the first of the problems with Eiffel's specification language described above.

9 Conclusions

Larch/C++ is a formal interface specification language, tailored to C++. As such one can precisely describe both the behavior and the C++ interface of parts of a C++ class library or module. Vendors that sell libraries of C++ classes, could use Larch/C++ to provide documentation that is precise as their source code, without resorting to giving the customers the source code.

In addition, Larch/C++ can be used to record implementation decisions, such as private members of a class. Although this feature can be abused, it allows Larch/C++ to be a useful tool for design as well as user-visible documentation.

The parts of Larch/C++ described above have evolved over the last few years, but are now relatively stable. A parser, documentation, and more examples are available from the authors by anonymous ftp from ftp.cs.iastate.edu. Work is progressing on a type checker and formal semantics.

Acknowledgements

Thanks to Jim Horning, Kevin Jones, Piero Colagrosso, Jacek Leszczylowski, Tapas Shomee, David Sims, Krishna Kishore Dhara, Yang Meng Tan, Al Baker, and Rob Lavey for comments on drafts. Thanks to Krishna Kishore Dhara and Al Baker for several discussions during the Formal Methods seminar of Summer 1992.

References


Tech Report: TR93-18a
Submission Date: June 14, 1994