This article appeared in *Functional Programming, Concurrency, Simulation and Automated Reasoning*, Peter E. Lauer (Ed.), Springer-Verlag Lecture Notes in Computer Science, Vol. 693, 1993, pages 329–348.

An Overview of Larch

Stephen J. Garland and John V. Guttag
MIT Laboratory for Computer Science

James J. Horning
Digital Equipment Corp. Systems Research Center

We begin by describing the Larch approach to specification and illustrating it with a few small examples. We then discuss LP, the Larch proof assistant, a tool that supports all the Larch languages. Our intent is to give you only a taste of these things. For a comprehensive look at Larch see [12].

Two-tiered specifications

The Larch family of languages supports a *two-tiered*, definitional style of specification. Each specification has components written in two languages: one language that is designed for a specific programming language and another language that is independent of any programming language. The former kind are *Larch interface languages*, and the latter is the *Larch Shared Language* (LSL).

Interface languages are used to specify the interfaces between program components. Each specification provides the information needed to use an interface. A critical part of each interface is how components communicate across the interface. Communication mechanisms differ from programming language to programming language. For example, some languages have mechanisms for signalling exceptional conditions, others do not. More subtle differences arise from the various parameter passing and storage allocation mechanisms used by different languages.

It is easier to be precise about communication when the interface specification language reflects the programming language. Specifications written in such interface languages are generally shorter than those written in a "universal" interface language. They are also clearer to programmers who use components and to programmers who implement them.

Each interface language deals with what can be observed by client programs written in a particular programming language. It provides a way to write assertions about program states, and it incorporates programming-language-specific notations for features such as side effects, exception handling, iterators, and concurrency. Its simplicity or complexity depends largely on the simplicity or complexity of its programming language.

Larch interface languages have been designed for a variety of programming languages. The two best developed languages are for C and for Modula-3. Other interface languages have been designed for Ada [2, 11], CLU [20], C++ [13, 21, 23], ML [24], and Smalltalk [3].

```
uses TaskQueue;
mutable type queue;
immutable type task;

task *getTask(queue q) {
  modifies q;
  ensures
    if isEmpty(q^)
      then result = NIL ∧ unchanged(q)
      else (*result)' = first(q^) ∧ q' = tail(q^);
}
```

Figure 1: An LCL interface specification

There are also "generic" Larch interface languages that can be specialized for particular programming languages or used to specify interfaces between programs in different languages [14, 22].

Larch interface languages encourage a style of programming that emphasizes the use of abstractions, and each provides a mechanism for specifying abstract types. If its programming language provides direct support for abstract types (as Modula-3 does), the interface language facility is modeled on that of the programming language; if its programming language does not (as C does not), the facility is designed to be compatible with other aspects of the programming language.

Figure 1 contains a sample interface specification for a small fragment of a scheduler for an operating system. The specification is written in LCL (a Larch interface language for C). This fragment introduces two abstract types and a procedure for selecting a task from a task queue. Briefly, `*` means pointer to (as in C), `result` refers to the value returned by the procedure, the symbol `^` is used to refer to the value in a location when the procedure is called, and the symbol `'` to refer to its value when the procedure returns.

The specification of `getTask` is not self-contained. For example, looking only at this specification there is no way to know which task `getTask` selects. Is it the one that has been in `q` the longest? Is it is the one in `q` with the highest priority?

Interface specifications rely on definitions from *auxiliary specifications*, written in LSL, to provide semantics for the primitive terms they use. Specifiers are not limited to a fixed set of notations, but can use LSL to define specialized vocabularies suitable for particular interface specifications or classes of specifications.

Figure 2 contains a portion of an LSL specification that specifies the operators used in the interface specification of `getTask`. Based on the information in this LSL specification, one can deduce that the task pointed to by the result of `getTask` is the most recently inserted `important` task, if such a task exists. Otherwise it is the most recently inserted task.

Many informal specifications have a structure similar to this. They implicitly rely on auxiliary specifications by describing an interface in terms of concepts with which readers are assumed to be familiar, such as sets, lists, coordinates, and windows. But they don't define these auxiliary concepts. Readers can misunderstand such specifications, unless their intuitive understanding exactly matches the specifier's. And there is no way to be sure that such intuitions do match. LSL specifications provide unambiguous mathematical definitions of the terms that appear in interface specifications.

```
TaskQueue: trait
  includes Nat
  task tuple of id: Nat, important: Bool
  introduces
    new: → queue
    __ ⊣ __: task, queue → queue
    isEmpty, hasImportant: queue → Bool
    first: queue → task
    tail: queue → queue
  asserts
    queue generated by new, ⊣
    ∀ t: task, q: queue
      isEmpty(new);
      ¬ isEmpty(t ⊣ q);
      ¬ hasImportant(new);
      hasImportant(t ⊣ q) ==
        t.important ∨ hasImportant(q);
      first(t ⊣ q) ==
        if t.important ∨ ¬ hasImportant(q)
          then t else first(q);
      tail(t ⊣ q) ==
        if first(t ⊣ q) = t then q else t ⊣ tail(q)
```

Figure 2: LSL specification used by getTask

Larch encourages a separation of concerns, with basic constructs in the LSL tier and programming details in the interface tier. We suggest that specifiers keep most of the complexity of specifications in the LSL tier for several reasons:

- LSL specifications are likely to be more reusable than interface specifications.
- LSL has a simpler underlying semantics than most programming languages (and hence than most interface languages), so specifiers are less likely to make mistakes, and any mistakes they do make are more easily found.
- It is easier to make and to check assertions about semantic properties of LSL specifications than about semantic properties of interface specifications.

Many programming errors are easily detected by running the program, that is, by testing it. While some Larch specifications can be executed, most of them cannot. The Larch style of specification emphasizes brevity and clarity rather than executability. To make it possible to validate specifications before implementing or executing them, Larch permits specifiers to make assertions about specifications that are intended to be redundant. These assertions can be checked mechanically. Several tools that assist specifiers in checking these assertions as they debug specifications are already in use, and others are under development.

LSL, The Larch Shared Language

LSL specifications define two kinds of symbols, *operators* and *sorts*. The concepts of operator and sort are the same as those used in conventional mathematical logic. They are

```
Table: trait
   includes Integer
   introduces
      new: → Tab
      add: Tab, Ind, Val → Tab
      __ ∈ __: Ind, Tab → Bool
      lookup: Tab, Ind → Val
      size: Tab → Int
   asserts ∀ i, i1: Ind, v: Val, t: Tab
        ¬(i ∈ new);
        i ∈ add(t, i1, v) == i = i1 ∨ i ∈ t;
        lookup(add(t, i, v), i1) ==
           if i = i1 then v else lookup(t, i1);
        size(new) == 0;
        size(add(t, i, v)) ==
            if i ∈ t then size(t) else size(t) + 1
```

Figure 3: `Table.lsl`

similar to the programming language concepts of procedure and type, but it is important not to confuse these two sets of concepts. When discussing LSL specifications, we will consistently use the words "operator" and "sort." When talking about programming language constructs, we will use the words "procedure" (or "function," "routine," or "method," as appropriate) and "type." Operators stand for total functions from tuples of values to values. Sorts stand for disjoint non-empty sets of values, and are used to indicate the domains and ranges of operators. In each interface language, "procedure" and "type" must mean what they mean in that programming language.

The *trait* is the basic unit of specification in LSL. A trait introduces some operators and specifies some of their properties. Sometimes the trait defines an abstract type. However, it is frequently useful to define a set of properties that does not fully characterize a type.

Figure 3 shows a trait that specifies a class of tables that store values in indexed places. It is similar to specifications in many "algebraic" specification languages [1, 4, 6, 25].

The specification begins by *including* another trait, `Integer`. This specification, which can be found in the LSL handbook in [12], supplies information about the operators `+`, `0`, and `1`, which are used in defining the operators introduced in `Table`.

The *introduces clause* declares a set of operators, each with its *signature* (the sorts of its domain and range). Signatures are used to sort-check terms in much the same way as procedure calls are type-checked in programming languages.

The *body* of the specification contains, following the reserved word `asserts`, equations between terms containing operators and variables.[1] The third equation resembles a recursive function definition, since the operator `lookup` appears on both the left and right sides. However, it merely states a relation that must hold among `lookup`, `add`, and the built-in operator `if_then_else_`; it does not fully define `lookup`. For example, it doesn't say anything about the value of the term `lookup(new, i)`.

Each well-formed trait defines a *theory* (a set of sentences closed under logical

[1]The equation connective in LSL, `==`, has the same semantics as the equality symbol, `=`. It is used only to introduce another level of precedence into the language.

consequence) in multisorted 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. This *loose* semantic interpretation guarantees that formulas in the theory follow only from the presence of assertions in the trait—never from their absence. This is in contrast to algebraic specification languages based on initial algebras [9] or final algebras [19]. Using the loose interpretation ensures that all theorems proved about an incomplete specification remain valid when it is extended.

Equational theories are useful, but a stronger theory is often needed, for example, when specifying an abstract type. The constructs `generated by` and `partitioned by` provide two ways of strengthening equational specifications.

A *generated by* clause asserts that a list of operators is a complete set of *generators* for a sort. That is, each value of the sort is equal to one that can be written as a finite number of applications of just those operators, and variables of other sorts. This justifies a *generator induction schema* for proving things about the sort. For example, the natural numbers are generated by `0` and `succ`, and the integers are generated by `0`, `succ`, and `pred`. The assertion

```
Tab generated by new, add
```

if added to `Table`, could be used to prove theorems by induction over `new` and `add`, since, according to this assertion, any value of sort `Tab` can be constructed from `new` by a finite number of applications of `add`.

A *partitioned by* clause asserts that a list of operators constitutes a complete set of *observers* for a sort. That is, all distinct values of the sort can be distinguished using just those operators. Terms that are not distinguishable using any of them are therefore equal. For example, sets are partitioned by $\in$, because sets that contain the same elements are equal. Each partitioned by clause is a new axiom that justifies a deduction rule for proofs about values of the sort. For example, the assertion

```
Tab partitioned by ∈, lookup
```

adds the deduction rule

```
∀ i1:ind (i1 ∈ t1 = i1 ∈ t2),
∀ i1:ind (lookup(t1, i1) = lookup(t2, i1)))
----------------------------
          t1 = t2
```

to the theory associated with `Table`.

It is instructive to note some of the things that `Table` does *not* specify:

1. It does not say how tables are to be represented.
2. It does not give algorithms to manipulate tables.
3. It does not say what procedures are to be implemented to operate on tables.
4. It does not say what happens if one looks up an `Ind` that is not in a `Tab`.

The first two decisions are in the province of the implementation. The third and fourth are recorded in interface specifications.

Interface specifications

An interface specification defines an interface between program components, and is written in a programming-language-specific Larch interface language. Each specification must provide the information needed to use an interface and to write programs that implement it. At the core of each Larch interface language is a model of the state manipulated by the associated programming language.

Program states

States are mappings from *locs* (abstract storage locations, also known as objects) to *values*. Each variable identifier has a type and is associated with a loc of that type. The major kinds of values that can be stored in locs are:

- *basic values*. These are mathematical constants, like the integer `3` and the letter `A`. Such values are independent of the state of any computation.
- *exposed types*. These are data structures that are fully described by the type constructors of the programming language (e.g., `int *` in C or `ARRAY [1..10] OF INTEGER` in Modula-3. The representation is visible to, and may be relied on by, clients.
- *abstract types*. Data types are best thought of as collections of related operations on collections of related values. Abstract types are used to hide representation information from clients.

Each interface language provides operators (e.g., ^ and ') that can be applied to locs to extract their values in the relevant states (usually the pre-state and the post-state of a procedure).

Each loc's type defines the kind of values it can map to in any state. Just as each loc has a unique type, each LSL term has a unique sort. To connect the two tiers in a Larch specification, there is a mapping from interface language types (including abstract types) to LSL sorts. Each type of basic value, exposed type, and abstract type is *based on* an LSL sort. Interface specifications are written using types and values. Properties of these values are defined in LSL, using operators on the corresponding sorts.

For each interface language, a standard LSL trait defines operators that can be applied to values of the sorts that the programming language's basic types and other exposed types are based on. Users familiar with the programming language will already have an intuitive understanding of these operators. Abstract types are typically based on sorts defined in traits supplied by specifiers.

Procedure specifications

The specification of each procedure in an interface can be studied, understood, and used without reference to the specifications of other procedures. A specification consists of a procedure header (declaring the types of its arguments and results) followed by a body of the form:

```
requires reqP
modifies modList
ensures  ensP
```

```
mutable type table;
uses Table(table for Tab, char for Ind,
           char for Val, int for Int);
constant int maxTabSize;

table table_create(void) {
  ensures result' = new ∧ fresh(result);
  }
bool table_add(table t, char i, char c) {
  modifies t;
  ensures result = (size(t^) < maxTabSize ∨ i ∈ t^)
    ∧ (if result then t' = add(t^, i, c)
                 else t' = t^);
  }
char table_read(table t, char i) {
  requires i ∈ t^;
  ensures result = lookup(t^, i);
  }
```

Figure 4: A Sample LCL Interface Specification

A specification places constraints on both clients and implementations of the procedure. The *requires clause* is used to state restrictions on the state, including the values of any parameters, at the time of any call. The *modifies* and *ensures clauses* place constraints on the procedure's behavior when it is called properly. They relate two states, the state when the procedure is called, the *pre-state*, and the state when it terminates, the *post-state*.

A requires clause refers only to values in the pre-state. An ensures clause may also refer to values in the post-state.

A modifies clause says what locs a procedure is allowed to change (its *target list*). It says that the procedure must not change the value of any locs visible to the client except for those in the target list. Any other loc must have the same value in the pre and post-states. If there is no modifies clause, then nothing may be changed.

For each call, it is the responsibility of the client to make the requires clause true in the pre-state. Having done that, the client may assume that:

- the procedure will terminate,
- changes will be limited to the locs in the target list, and
- the postcondition will be true on termination.

The client need not be concerned with how this happens.

The implementor of a procedure is entitled to assume that the precondition holds on entry, and is only responsible for the procedure's behavior if it is. A procedure's behavior is totally unconstrained if its precondition isn't satisfied, so it is good style to keep the requires clause weak. An omitted requires clause is equivalent to `requires true` (the weakest possible requirement).

```
INTERFACE Table;
<* TRAITS Table(CHAR FOR Ind, CHAR FOR Val,
                INTEGER FOR Int) *>
  TYPE T <: OBJECT
    METHODS
      Add(i: CHAR; c: CHAR) RAISES {Full};
      Read(i: CHAR): CHAR;
    END;
  PROCEDURE Create( ): T;
  CONST MaxTabSize: INTEGER = 100;
  EXCEPTION Full;
<*
  FIELDS OF T
    val :  Tab;
  METHOD T.Add(i, c)
    MODIFIES SELF.val
    ENSURES SELF.val' = add(SELF.val, i, c)
      EXCEPT size(SELF.val) ≥ MaxTabSize
          ∧ ¬(i ∈ SELF.val)
        => RAISEVAL = Full ∧ UNCHANGED(ALL)
  METHOD T.Read(i)
    REQUIRES i ∈ SELF.val
    ENSURES RESULT = lookup(SELF.val, i)
  PROCEDURE Create
    ENSURES RESULT.val = new ∧ FRESH(RESULT)
*>
END Table.
```

Figure 5: A Sample LM3 Interface Specification

```
void choose(int x, int y) int z; {
  modifies z;
  ensures z' = x ∨ z' = y;
}
```

Figure 6: A specification of choose

```
void choose(int x, int y) {
  if (x > y) z = x;
  else z = y;
}
```

Figure 7: An implementation of choose

Two interface language examples

Figure 4 contains a fragment of a specification written in LCL (a Larch interface language for Standard C). Figure 5 contains a fragment of a similar specification written in LM3 (a Larch interface language for Modula-3). They use the same `Table` trait of Figure 3. We present these examples here simply to convey an impression of how programming language dependencies influence Larch interface languages. The notations used are described in detail in [12]

Relating implementations to specifications

In our work we emphasize using specifications as a communication medium. Programmers are encouraged to become clients of well-specified abstractions that have been implemented by others.

One of the advantages of Larch's two-tiered approach to specification is that the relationship of implementations to specifications is relatively straightforward. Consider, for example, the LCL specification in Figure 6 and the C implementation in Figure 7.

The specification defines a relation between the program state when `choose` is called and the state when it returns. This relation contains all pairs of states <*pre, post*> in which

- the states differ only in the value of the global variable `z`, and
- in *post* the value of `z` is that of one of the two arguments passed to `choose`.

The implementation also defines a relation on program states. This relation contains all pairs of states <*pre, post*> in which

- the states differ only in the value of the variable `z`, and
- in *post* the value of `z` is the maximum of the two arguments passed to `choose`.

We say that the implementation of `choose` in Figure 7 *satisfies* the specification in

Figure 6—or is a *correct implementation* of Figure 6[2]—because the relation defined by the implementation is a subset of the relation defined by the specification. Every possible behavior that can be observed by a client of the implementation is permitted by the specification.

The definition of satisfaction we have just given is not directly useful. In practice, formal arguments about programs are not usually made by building and comparing relations. Instead, such proofs are usually done by pushing predicates through the program text, in ways that can be justified by appeal to the definition of satisfaction. A description of how to do this appears in the books [5, 10].

The notion of satisfaction is a bit more complicated for implementations of abstract types, because the implementor of an abstract type is working simultaneously at two levels of abstraction. To implement an abstract type, one chooses data structures to represent values of the type, then writes the procedures of the type in terms of that representation. However, since the specifications of those procedures are in terms of abstract values, one must be able to relate the representation data structures to the abstract values that they represent. This relation is an essential (but too often implicit) part of the implementation.

Figure 8 shows an implementation of the LCL specification in Figure 4. A value of the abstract type `table` is represented by a pointer to a struct containing two arrays and an integer. You need not look at the details of the code to understand the basic idea behind this implementation. Instead, you should consider the abstraction function and representation invariant.

The *abstraction function* is the bridge between the data structure used in the implementation of an abstract type and the abstract values being implemented. It maps each value of the representation type to a value of the abstract type. Here, we represent a `table` by a pointer, call it `t`, to a struct. If the triple `<ind, val, next>` contains the values of the fields of that struct in some state `s`, then we can define the abstract value represented by `t` in state `s` as `toTab(<ind, val, next>)`, where

```
toTab(<next, ind, val>) ==
  if next = 0 then empty
  else insert(toTab(<next - 1, ind, val>),
              ind[next], val[next])
```

Abstraction functions are often many-to-one. Here, for example, if `t->next = 0`, `t` represents the empty `table`, no matter what the contents of `t->ind` and `t->val`.

The typedefs in Figure 8 define a data structure sufficient to represent any value of type `table`. However, it is not the case that any value of that data structure represents a value of type `table`. In defining the abstraction function, we relied upon some implicit assumptions about which data structures were valid representations. For example, `toTab` is not defined when `t->next` is negative. A *representation invariant* is used to make such assumptions explicit. For this implementation, the representation invariant is

- The value of `next` lies between `0` and `maxTabSize`:

  $0 \leq$ `t->next` $\wedge$ `t->next` $\leq$ `maxTabSize`

[2] "Correct" is a dangerous word. It is not meaningful to say that an implementation is "correct" or "incorrect" without saying what specification it is claimed to satisfy. The technical sense of "correct" that is used in the formal methods community does not imply "good," or "useful," or even "not wrong," but merely "consistent with its specification."

```
#include "bool.h"
#define maxTabSize (10)

typedef struct {char ind[maxTabSize];
                char val[maxTabSize];
                int next;} tableRep;
typedef tableRep * table;

table table_create(void) {
  table t;
  t = (table) malloc(sizeof(tableRep));
  if (t == 0) {
    printf("Malloc returned null in table_create\n");
    exit(1);
  }
  t->next = 0;
  return t;
}
bool table_add(table t, char i, char c) {
  int j;
  for (j = 0; j < t->next; j++)
    if (t->ind[j] == i) {
      t->val[j] = c;
      return TRUE;
    }
  if (t->next == maxTabSize) return FALSE;
  t->val[t->next++] = c;
  return TRUE;
}
char table_read(table t, char i) {
  int j;
  for (j = 0; TRUE; j++)
    if (t->ind[j] == i) return t->val[j];
}
```

Figure 8: Implementing an Abstract Type

- and no index may appear more than once in the fragment of `ind` that lies between `0` and `next`:

```
∀ i,j:int
  (0 ≤ i ∧ i < j ∧ j < t->next)
    ⇒ (t->ind)[i] ≠ (t->ind)[j]
```

To show that that this representation invariant holds, we use a proof technique called *data type induction*. Since `table` is an abstract type, we know that clients cannot directly access the data structure used to represent a `table`. Therefore, all values of type `table` that occur during program execution will have been generated by the functions specified in the interface. So to show that the invariant holds it suffices to show, reasoning from the code implementing the functions on `table`s, that

- the value returned by `table_create` satisfies the invariant (this is the basis step of the induction),
- whenever `table_add` is called, if the invariant holds for `t^` then the invariant will also hold for `t'`, and
- whenever `table_read` is called, if the invariant holds for `t^` then the invariant will also hold for `t'`.

A slightly different data type induction principle can be used to reason about clients of abstract types. To prove that a property holds for all instances of the type, i.e., that is is an *abstract invariant*, one inducts over all possible sequences of calls to the procedures that create or modify locs of the type. However, one reasons using the specifications of the procedures rather than their implementations. For example, to show that the `size(t)` is never greater than `maxTabSize` one shows that

- the specification of `table_create` implies that the size of the `table` returned is not greater than `maxTabSize`, and
- the specification of `table_add` combined with the hypothesis `t^` ≤ `maxTabSize` implies that `t'` ≤ `maxTabSize`.

Given the abstraction function, it is relatively easy to define what it means for the procedure implementations in Figure 8 to satisfy the specifications in Figure 4. For example, we say that the implementation of `table_read` satisfies its specification because the image under the abstraction function of the relation between pre and post-states defined by the implementation (i.e., what one gets by applying the abstraction function to all values of type `table` in the relation defined by the implementation) is a subset of the relation defined by the specification. Notice, by the way, that any argument that the implementation of `table_read` satisfies its specification will rely on both the `requires` clause of the specification and on the representation invariant.

LP, The Larch Proof Assistant

The discussions of LSL, LCL, and LM3 have alluded to tools supporting those languages. LP is a tool that is used to support all three. It is described in [8]. Here we give merely a glimpse of its use.

LP is a proof assistant for a subset of multisorted first-order logic with equality, the logic on which the Larch languages are based. It is designed to work efficiently on large problems and to be used by specifiers with relatively little experience with theorem proving. Its design and development have been motivated primarily by our work on LSL, but it also has other uses, for example, reasoning about circuit designs [17, 18], algorithms involving concurrency [7], data types [23], and algebraic systems [15].

LP is intended primarily as an interactive proof assistant or proof debugger, rather than as a fully automatic theorem prover. Its design is based on the assumption that initial attempts to state and prove conjectures usually fail. So LP is designed to carry out routine (but possibly lengthy) proof steps automatically and to provide useful information about why proofs fail. To keep users from being surprised and confused by its behavior, LP does not employ complicated heuristics for finding proofs automatically. It makes it easy for users to employ standard techniques such as proof by cases, by induction, or by contradiction, but the choice among such strategies is left to the user.

The life cycle of proofs

Proving is similar to programming: proofs are designed, coded, debugged, and (sometimes) documented.

Before designing a proof it is necessary to formalize the things being reasoned about and the conjecture to be proved. The design of the proof proper starts with an outline of its structure, including key lemmas and methods of proof. The proof itself must be given in sufficient detail to be convincing. What it means to be convincing depends on who (or what) is to be convinced. Experience shows that humans are frequently convinced by unsound proofs, so we look for a mechanical "skeptic" that is just hard enough (but not too hard) to convince.

Once part of a proof has been coded, LP can be used to debug it. Proofs of interesting conjectures hardly ever succeed the first time. Sometimes the conjecture is wrong. Sometimes the formalization is incorrect or incomplete. Sometimes the proof strategy is flawed or not detailed enough. LP provides a variety of facilities that can be used to understand the problem when an attempted proof fails.

While debugging proofs, users frequently reformulate axioms and conjectures. After any change in the axiomatization, it is necessary to recheck not only the conjecture whose proof attempt uncovered the problem, but also the conjectures previously proved using the old axioms. LP has facilities that support such regression testing.

LP will, upon request, record a session in a script file that can be replayed. LP "prettyprints" script files, using indentation to reflect the structure of proofs. It also annotates script files with information that indicates when subgoals are introduced (e.g., in a proof by induction), and when subgoals and theorems are proved. On request, as LP replays a script file, it will halt replay at the first point where the annotations and the new proof diverge. This checking makes it easier to keep proof attempts from getting "out of sync" with their author's conception of their structure.

A small proof

Figure 9 contains a short LSL specification, including a simple conjecture (following the reserved word `implies`) that is supposed to follow from the axioms. Figure 10 shows a script for an LP proof of that conjecture.

```
Nat: trait
  includes AC(+, Nat)
  introduces
    0: → Nat
    s: Nat → Nat
    __ < __: Nat, Nat → Bool
  asserts
    Nat generated by 0, s
    ∀ i, j, k: Nat
      i + 0 == i;
      i + s(j) == s(i + j);
      ¬(i < 0);
      0 < s(i);
      s(i) < s(j) == i < j
    implies ∀ i, j, k: Nat
      i < j ⇒ i < (j + k)
```

Figure 9: A trait containing a conjecture

```
set name nat
declare sort Nat
declare variables i, j, k: Nat
declare operators
  0: → Nat
  s: Nat → Nat
  +: Nat, Nat → Nat
  <: Nat, Nat → Bool
  ..
assert Nat generated by 0, s
assert ac +
assert
  i + 0 == i
  i + s(j) == s(i + j)
  ¬(i < 0)
  0 < s(i)
  s(i) < s(j) == i < j
  ..
set name lemma
prove i < j ⇒ i < (j + k) by induction on j
  <> 2 subgoals for proof by induction on j
    [] basis subgoal
    resume by induction on i
    <> 2 subgoals for proof by induction on i
      [] basis subgoal
      [] induction subgoal
    [] induction subgoal
  [] conjecture
qed
```

Figure 10: Sample LP proof script

The `declare` commands introduce the variables and operators in the LSL specification. The `assert` commands supply the LSL axioms relating the operators; the `Nat generated by` assertion provides an induction scheme for `Nat`. The `prove` command initiates a proof by induction of the conjecture. The *diamond* (`<>`) annotations are provided by LP; they indicate the introduction of subgoals for the inductions. The *box* (`[]`) annotations are also provided by LP; they indicate the discharge of subgoals and, finally, of the main proof. The `resume` command starts a nested induction. No other user intervention is needed to complete this proof. The `qed` command on the last line asks LP to confirm that there are no outstanding conjectures.

Conclusion

Larch is still very much a "work in progress." New Larch interface languages are being designed, new tools are being built, and the existing languages and tools are in a state of evolution. Most significantly, specifications are being written.

But Larch has reached a divide, what Churchill might have called "the end of the beginning." Until now, most of the work on Larch has been done by the authors of this paper and their close associates. We hope that the First International Workshop on Larch [16] and the publication of [12] mark the beginning of the period when most Larch research, development, and application will be done by people we do not yet know.

References

[1] Michel Bidoit. *Pluss, un langage pour le développement de spécifications algébriques modulaires.* Thèse d'Etat, Université de Paris-Sud, Orsay, May 1989.

[2] S.R. Cardenas and H. Oktaba. *Formal Specification in Larch Case Study: Text Manager. Interface Specification, Implementation, in Ada and Validation of Implementation*, TR 511, Instituto de Investigaciones en Matematicas Aplicadas y en Sistemas, Universidad Nacional Autonoma de Mexico, 1988.

[3] Yoonsik Cheon. *Larch/Smalltalk: A Specification Language for Smalltalk*, M.Sc. Thesis, Iowa State University, 1991.

[4] O.-J. Dahl, D.F. Langmyhr, and O. Owe. *Preliminary Report on the Specification and Programming Language ABEL*, Research Report 106, Institute of Informatics, University of Oslo, Norway, 1986.

[5] Ole-Johan Dahl. *Verifiable Programming*, Prentice Hall International Series in Computer Science, 1992.

[6] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, EATCS Monographs on Theoretical Computer Science, vol. 6, Springer-Verlag, 1985.

[7] Urban Engberg, Peter Grønning, and Leslie Lamport. "Mechanical verification of concurrent systems with TLA," *Proc. Workshop on Computer Aided Verification*, 1992. Revised version in [16].

[8] Stephen J. Garland and John V. Guttag. *A Guide to LP, The Larch Prover*, TR 82, DEC/SRC, Dec. 1991.

[9] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. "An initial algebra approach to the specification, correctness and implementation of abstract data types," *Current Trends in Programming Methodology IV: Data Structuring*, R. Yeh (ed.), Prentice-Hall, 1978.

[10] David Gries. *The Science of Programming*, Springer-Verlag, 1981.

[11] David Guaspari, Carla Marceau, and Wolfgang Polak. "Formal verification of Ada," *IEEE Trans. Software Engineering* 16(9), Sept. 1990.

[12] J.V. Guttag and J.J. Horning (eds.) with S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing *Larch, Languages and Tools for Formal Specification*, Springer-Verlag, 1993 (to appear).

[13] Gary T. Leavens and Yoonsik Cheon. "Preliminary design of Larch/C++," in [16].

[14] Richard Allen Lerner. *Specifying Objects of Concurrent Systems*, Ph.D. Thesis, Dept. of Computer Science, Carnegie Mellon University, TR CS-91-131, May 1991.

[15] U. Martin and T. Nipkow. "Automating Squiggol," *Proc. IFIP Work. Conf. Programming Concepts and Methods*, Tiberias, Apr. 1990. North-Holland.

[16] U. Martin and J.M. Wing. *Proc. First Intl. Workshop on Larch*, Dedham, Jul. 1992, Springer-Verlag.

[17] James B. Saxe, Stephen J. Garland, John V. Guttag, and James J. Horning. "Using Transformations and Verification in Circuit Design," in [16].

[18] Jørgen Staunstrup, Stephen J. Garland, and John V. Guttag. "Mechanized verification of circuit descriptions using the Larch Prover," *Proc. IFIP Work. Conf. Theorem Provers in Circuit Design: Theory, Practice, and Experience*, Nijmegen, Jun. 1992. North-Holland.

[19] M. Wand. "Final algebra semantics and data type extensions," *Journal of Computer and System Sciences*, Aug. 1979.

[20] Jeannette Marie Wing. *A Two-Tiered Approach to Specifying Programs*, Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, MIT, TR MIT/LCS/TR-299, May 1983.

[21] J.M. Wing. "Using Larch to Specify Avalon/C++ Objects," *Proc. Intl. Joint Conf. Theory and Practice of Software Development, TAPSOFT*, Barcelona, Mar. 1989. Springer-Verlag, LNCS 352. Revised version in [21].

[22] Jeannette M. Wing. "Writing Larch Interface Language Specifications," *ACM Trans. Programming Languages and Systems* 9(1), Jan. 1987.

[23] Jeannette M. Wing and Chun Gong. "Experience with the Larch Prover," *Proc. ACM Intl. Workshop on Formal Methods in Software Development*, May 1990.

[24] J.M. Wing, Eugene Rollins, and Amy Moormann Zaremski. "Thoughts on a Larch/ML and a new application for LP," in [16].

[25] M. Wirsing. *Algebraic Specification*, Technical Report MIP-8914, University of Passau, Germany, 1989.