Thoughts on a Larch/ML and a New Application for LP

Jeannette M. Wing, Eugene Rollins, and Amy Moormann Zaremski

May 7, 1992
CMU-CS-92-135

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in *The First International Workshop on Larch*, July 1992.

Abstract

We describe a preliminary design for a Larch interface language for the programming language ML.[1] ML's support for higher-order functions suggests a need to go beyond the first-order logical foundations of Larch languages. We also propose a new application, *specification matching*, for the Larch Prover, which could benefit from extending LP to handle full first-order logic. This paper describes on-going work and suggests a number of open problems related to Larch/ML and to LP as used for specification matching. We assume rudimentary knowledge of Larch, its languages and two-tiered approach.

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597.

Keywords: formal methods, formal specification, specification matching, Larch, Larch Prover, Standard ML

1 Introduction

1.1 ML

ML is a "mostly-functional," strongly-typed, polymorphic programming language with a published formal semantics [8]. On top of ML's small core language is a *module* facility that supports the incremental construction of large programs. An ML program is a collection of self-contained program units with explicitly-specified interfaces. ML programmers define interfaces, called *signatures*, separate from their implementations, called *structures*. Structures define environments (*i.e.*, bindings between identifiers and values); they are ML's way of encapsulating sets of variable, function, and type definitions. *Functors* are parameterized structures and are used to create and compose structures: application of a functor to a structure that matches a given signature yields a structure. ML supports separate compilation through its ability to export and import functors.

A Larch/ML specification extends the information contained in an ML signature. Like other Larch interface specifications, it associates ML types with Larch Shared Language (LSL) sorts and uses pre- and post-conditions to specify the behavior of declared functions. It adds semantic information to the syntactic information contained in a signature, but abstracts from the details of data representation and specific algorithms that would be found in the structures or functors that implement it.

1.2 Why Larch and ML?

Larch complements ML. The Larch and ML communities both advocate taking a rigorous approach to software development. Larch is particularly well-suited for specifying properties of data types. Whereas precise static and dynamic semantics have been given for ML, they have assumed a set of basic types from which more interesting data types are built. Semantic properties of even the basic types in ML are left unspecified; hence, Larch provides a way to specify these properties.

Though most ML programmers strive to write code in a purely functional style, the ML community is not dogmatic about it. For example, in the Venari Project at Carnegie Mellon we are addressing issues like concurrency and persistence, and have found the existence of "impure" features critical, not just for performance, but for expressibility (*e.g.*, the Threads interface to ML exports sharable mutex locks and adds a notion of process state [1]). Larch addresses state changes like side effects and resource allocation (*e.g.*, creation of new objects) explicitly in interfaces. Larch interfaces highlight potential state changes that an ML programmer might otherwise miss when reading just the code.

Since Larch interfaces are defined in terms of the target programming language's model of state, to define a Larch/ML formally one needs to write an ML "state" trait. Writing such a "state" trait has always been a difficult exercise in designing other Larch interface languages since most programming languages do not come equipped with a formal semantics and/or their model of state is complex. Fortunately ML has both a formal semantics and a straightforward notion of state, so writing the Larch traits for ML state should be simpler than doing so for other programming languages (like C).

In Section 2 we give two examples of Larch/ML interface specifications and discuss some open problems.

1.3 A New Application for LP: Specification Matching

We are working on the design of Larch/ML in the context of the Venari Project's application of interest: searching large software libraries. We would like to be able to use specifications as search keys and do *specification matching* to determine whether a module should be returned as a result of a query [14]. We assume a specification, s_i, is associated with each component, p_i, in a software library of i components. For example, a procedure's specification might be a Larch interface describing the procedure's behavior. Specification matching is the process of determining whether for a given query, q, and specification, s_i, s_i *satisfies* q. If we assume the query and specification language are both drawn from the same logical language then *satisfies* is logical implication; specification matching is the process of showing an implication holds.

In Section 3 we discuss how we might use the Larch Prover as the backend "theorem prover" to do specification matching and suggest some extensions to LP that would make our task easier.

2 Larch/ML Examples

2.1 Queue

2.1.1 A Specification

Figure 1 shows an example Larch/ML specification of a queue type. We elaborate the ML signature module, QUEUE, with Larch/ML specification information, delimited by (∗+. . .+∗). Since ML comments are delimited by (∗ . . . ∗), we can compile a Larch/ML specification as a normal signature, using the ML compiler without preprocessing.

The specification begins with a **using** clause, which lists traits used by the Larch/ML interface. QUEUE uses the Que trait (Figure 2). In using several traits, name conflicts may arise between these traits, or names in the signature. In most Larch interface languages, one must do renaming to resolve the conflicts. We chose to use ML's mechanism of qualified identifiers, using trait names to qualify operator names.

A **based on** clause may be attached to any type declaration to associate ML types with LSL sorts. In Figure 1, the **based on** clause associates the ML type t with the sort C from the Que trait. It also states that in any use of the type constructor t, the instance for the type variable 'a must be associated with a sort that can substitute for the sort E from the Que trait.[1] Later we explain how we type check specifications.

[1] In ML, type variables like 'a begin with a prime and are used for defining polymorphic types.

```
signature QUEUE = sig
  (*+ using Que +*)
  type 'a t (*+ based on Que.E Que.C +*)
  exception Deq (*+ raised by deq +*)
  val create: unit → 'a t
     (*+ create ( ) = q
        ensures q = Que.new +*)
  val enq: 'a t * 'a → 'a t
     (*+ enq (q,e)
        ensures result = Que.insert(q,e) +*)
  val deq: 'a t → ('a * 'a t)
     (*+ deq q = (e, q1)
        ensures if Que.isEmpty(q) then raise Deq
                else (e = Que.first(q)) and (q1 = Que.rest(q)) +*)
  val len: 'a t → int
     (*+ len q
        ensures result = Que.size(q) +*)
  end
```

Figure 1: Larch/ML Specification of an Immutable Queue

An ML signature declares exception values that may be raised in the module; the **raised by** clause enables specifiers to name the functions that may raise a particular exception. In QUEUE, Deq is the only exception value and deq is the only function that may raise it.

The specification for each function begins with a *call pattern* consisting of the function name followed by a pattern for each parameter, optionally followed by an equal sign (=) and a pattern for the result. In ML, patterns are used in binding constructs to associate names to parts of values (*e.g.*, (x, y) names x as the first of a pair and y as the second). ML programmers typically use patterns in function implementations to bind names to parameters and subcomponents of parameters; we borrow this ML feature as a way to introduce names in specifications. We adopt the convention that if the result pattern is not given, the default name for the result value is "result" (*e.g.*, as used in the enq and len functions in QUEUE).

Like other Larch interfaces, the **requires** clause specifies the function's pre-condition as a predicate consisting of trait operators and names introduced by the call pattern. Similarly, an **ensures** clause specifies the function's post-condition. If a function does not have an explicit **requires** clause, the default is **requires true**. In the QUEUE example, none of the functions have explicit **requires** clauses.

In QUEUE, the create function returns an empty queue; enq returns a queue that

```
Que(E,C) : trait
  includes Integer
  introduces
    new: → C
    insert: C, E → C
    first: C → E
    rest: C → C
    isEmpty: C → Bool
    size: C → Int
  asserts
    C generated by new, insert
    ∀ q: C, e: E
      first(insert(q, e)) == if isEmpty(q) then e else first(q)
      rest(insert(q, e)) ==
        if isEmpty(q) then new else insert(rest(q), e)
      isEmpty(new)
      ~ isEmpty(insert(q, e))
      size(new) == 0
      size(insert(q,e)) == succ(size(q))
```

Figure 2: Que Trait

is a result of inserting an element in a given queue; deq raises the exception Deq if the queue is empty and otherwise returns a tuple whose respective values are the first element of the queue and the rest of the queue; finally, len returns the size of a given queue.

2.1.2 Type-checking

In type-checking a function specification, we assign the names in the call pattern ML types based on the type of the function. These ML types are associated with LSL sorts according to the **based on** clauses (or built-in associations for base types). We type check the **requires** and **ensures** clauses using the sorts associated with the types of the pattern names and the signatures of trait operators and interpreting the resulting type expression. For example, suppose the following were part of the QUEUE specification in Figure 1.

```
  val convert: int t → bool t
    (*+ convert q
      ensures Que.first(q) = Que.first(result) +*)
```

This function specification contains a type error, discovered upon type checking: In step 1, we list the names used in the **ensures** predicate, along with their types, and substitute (⇒) associated sorts for ML types. In step 2, we substitute sort expressions for names within the **ensures** predicate, then simplify in step 3 and note the type error.

1)	q	: int t	(substitute sorts)	⇒ Int C
	result	: bool t	(substitute sorts)	⇒ Bool C
	first	: (E C) → E		

2) ((E C) → E) (Int C) = ((E C) → E) (Bool C)

3) Int = Bool

This next example converts a queue to a stack. It shows how type checking works in the presence of type variables and multiple type constructors; it uses sorts from two different traits, Que and Stk.

signature StackToQueue = **sig**
(∗+ **using** Que, Stk +∗)
type 'a t (∗+ **based on** Que.E Que.C +∗)
type 'a s (∗+ **based on** Stk.A Stk.S +∗)
val convert: 'a t → 'a s
    (∗+ convert q
        ensures Que.first(q) = Stk.top(result) +∗)

This example type checks successfully. In the second step below, the type checker unifies E of (E C) with the 'a of ('a C), which implies that any type that substitutes for 'a must be associated with a sort that can substitute for E in Que; the A of (A S) unifies with the 'a of ('a S), with a similar implication.

1)	q	:'a t	(substitute sorts)	⇒ 'a C
	result	:'a s	(substitute sorts)	⇒ 'a S
	first	: (E C) → E		
	top	: (A S) → A		

2) ((E C) → E) ('a C) = ((A S) → A) ('a S)

3) 'a = 'a

The expression Que.first(result) would not type check. Substituting sort expressions gives us ((E C) → E) ('a S), but (E C) does not unify with ('a S).

2.1.3 Example Implementations

Figures 3 and 4 show two different ML implementations that satisfy the QUEUE specification. The first implementation represents a queue as a list, where the head of the list is the first element. The second implementation represents a queue as a pair of lists, where the head of the first list is the first element and the head of the second list is the last element. If a dequeue is performed when the first list is empty, the second list is reversed to form a new first list.

```
structure Queue : QUEUE =
  struct
    type 'a t = 'a list

    exception Deq

    fun create ( ) = [ ]

    fun enq ([ ],x) = [x]
      | enq ((hd::tl),x) = hd::(enq(tl,x))

    fun deq [ ] = raise Deq
      | deq (hd::tl) = (hd,tl)

    fun len q = length q

  end
```

Figure 3: A Queue Implemented by a List

```
structure Queue : QUEUE =
  struct
    datatype 'a t = Q of {front: 'a list, rear: 'a list}

    exception Deq

    fun create () = Q{front=nil,rear=nil}

    fun enq ((Q{front=f,rear=r}),x) = Q{front=f,rear=(x::r)}

    fun deq (Q{front=(hd::tl),rear=r}) = (hd,Q{front=tl,rear=r})
      | deq (Q{front=nil,rear=nil}) = raise Deq
      | deq (Q{front=nil,rear=r}) = deq(Q{front=rev r,rear=nil})

    fun len (Q {rear,front}) = length (rear) + length (front)
  end
```

Figure 4: A Queue Implemented by a Pair of Lists

2.2 Symbol Table

Although most ML programs are purely functional, ML programmers do use imperative constructs (*i.e.*, refs, arrays, and commands for input and output). Thus, we do need to be able to specify side effects. Larch is particularly appropriate for dealing with the impure aspects of ML; in contrast, Sannella and Tarlecki's Extended ML specification language [11], also based on algebraic axioms, handles only the purely functional subset of ML.

The symbol table example in Figure 5 shows a Larch/ML interface used to specify a mutable symbol table. It uses the SymTab trait in Figure 6.

The **based on** clause states that objects of the ML type table range over values denoted by the terms of sort S specified in the SymTab trait. Each **modifies** clause lists those objects whose value may possibly change as a result of executing the corresponding function; lookup is not allowed to change the state of its symbol table argument, but insert and delete are. In an **ensures** clause we use t% to stand for the value of the table in the final state and simply t for the value in the initial state. The clause **fresh**(t_{obj}) in the init function specifies that the object t_{obj} is newly created.

This example shows how in ML a signature may contain substructures, e.g., Key and Value, and hence refer to names they export, e.g., key and value. For this reason, to avoid possible name conflicts in the pre- and post-conditions, we qualify each operator name with the name of the trait in which it is introduced; alternatively, we could name for each trait listed in the **using** clause the sort and operator names explicitly "imported" from that trait.

2.3 Open Problems

2.3.1 Handling higher-order functions

ML functions typically take functions as arguments and return them as results. Since the assertion language of Larch interfaces is first-order, we cannot write in the post-condition of function P the assertion "apply(Q, A)" where "apply" is a higher-order function, Q is the functional argument to P and A is some list of arguments for Q. We do not necessarily even want to state this kind of assertion since the interface should specify only the effects of P, not how to achieve them (*e.g.*, by applying Q). In principle, since P's implementor is not required to call Q, but only to ensure P's effects are as if P called Q, P's implementor is free to completely ignore Q.

Our current design of Larch/ML mimics other Larch interface languages by handling functional arguments using a "macro-substitution" approach. Instead of writing "apply(Q, A)" we would write in P's post-condition something like "Q.spec **with** (A **for** F)," which refers to Q's specification, Q.spec, with appropriate renamings of actuals, A, for formals, F.

For example, in Figure 7, the function map takes a function convert as an argument. The specification of map refers to the specification of convert as convert.spec. We can think of these references as follows. For each application of map, the specification

```
signature SYMBOLTABLE = sig
  (*+ using SymTab +*)
  structure Key: KEY
  structure Value: VALUE
  type table (*+ based on SymTab.S +*)
  val init: unit → table
     (*+ init () = t
        ensures t = SymTab.emp and fresh(t_obj) +*)
  val insert: table * Key.key * Value.value → unit
     (*+ insert (t, k, v)
        modifies t
        ensures t% = SymTab.add(t, k, v) +*)
  val lookup: table * Key.key → Value.value
     (*+ lookup (t, k) = v
        requires SymTab.isin(t, k)
        ensures v = SymTab.find(t, k) +*)
  val delete: table * Key.key → unit
     (*+ delete (t, k)
        requires SymTab.isin(t, k)
        modifies t
        ensures t% = SymTab.rem(t, k) +*)
  end
```

Figure 5: Larch/ML Specification of a Mutable Symbol Table

```
SymTab: trait
  introduces
     emp: → S
     add: S, K, V → S
     rem: S, K → S
     find: S, K → V
     isin: S, K → Bool
  asserts
     S generated by (emp, add)
     S partitioned by (find, isin)
     ∀ (s: S, k, k1: K, v: V)
       rem(add(s, k, v), k1) == if k = k1 then rem(s, k)
         else add(rem(s, k1), k, v)
       find(add(s, k, v), k1) == if k = k1 then v else find(s, k1)
       isin(emp, k) == false
       isin(add(s, k, v), k1) == (k = k1) ∨ isin(s, k1)
  implies
     converts (rem, find, isin) exempting ∀ k: K (rem(emp, k), find(emp, k))
```

Figure 6: SymTab Trait

clause of the actual parameter for convert is macro-expanded into the specification of map. In the call pattern for map, we give the argument of convert a local name, x, which stands for the argument of any function passed as an actual parameter for convert. Renaming of the local name stands for renaming of the argument within the specification of any function passed as an actual parameter for convert. The local name result is given to the result of all function parameters. Note that the specification of map refers to itself. If we assume that macro expansion is lazy, a recursively-defined specification is well-formed if a base case is given.

```
signature Q = sig
  (*+ using Que +*)
  type 'a t (*+ based on Que.E Que.C +*)
  val map: ('a → 'b) → 'a t → 'b t
      (*+ map (convert(x)) q
          ensures
              convert.spec with
                  (Que.first(q) for x, Que.first(result) for result)
          and
              map.spec with
                  (convert for convert, Que.rest(q) for q,
                  Que.rest (result) for result) +*)
end
```

Figure 7: Specification of a Higher-Order Function

This solution is not entirely satisfying. Aside from the cumbersome syntax,[2] the thorniest technical problem is in dealing with functional arguments with side effects. Since an interface denotes a predicate over two states, in specifying the behavior of a function P with a side-effecting functional argument Q, we cannot explicitly assert anything about the intermediate states that P would see in calling Q, but be hidden from P's caller. The semantics of Larch interface languages designed so far (*e.g.*, for C [3], CLU [13], and Modula-3 [6]) have either ignored or have been limiting in their way of handling functional arguments. The advantage of looking at this issue in ML is that ML's formal semantics makes it possible to talk precisely about state and side effects. On the flip side, ML naturally pushes Larch to deal with higher-order functions as "first-class citizens" rather than as at the fringes of more traditional imperative programming languages; this push may be sufficient impetus for going beyond Larch's first-order restriction.

[2] which would be especially annoying to ML programmers who use functional arguments as the norm rather than the exception.

2.3.2 Sharing

In ML it is possible, and sometimes necessary, to allow sharing of types and even substructures between different program modules. For example, in building a compiler, a parser module and an abstract syntax tree module might need to share a symbol table. ML has a facility for letting programmers specify *sharing constraints* between types and between structures. Checking whether a constraint is met is determined by type or structure equality, as the case may be, since this check is easily computable. In a more complete Larch/ML we would like to allow the specifier to attach additional semantic constraints (*e.g.*, expressed as equations or a trait) on a shared type or structure. We would view these constraints as a shared theory in the sense of Larch.

2.3.3 Polymorphism

ML functions and structures can be defined using type variables. The closest that previous Larch interface languages have come to dealing with type variables is with CLU's parameterized modules. Again, the Larch treatment was syntactic in nature. Unlike for functional arguments, however, the syntactic approach to dealing with type variables should suffice for a Larch/ML, which we have done in our current design. However, we need to do a more careful semantic analysis to support this claim.

2.3.4 Other "types" of types

Features of ML we have not touched on in this paper include: *equality types*, *strong* versus *weak* types, **datatype**s, and **abstraction**s. We have not at all addressed the semantics of Larch/ML with respect to these features; some (equality and weak types) raise new questions and others (datatypes and abstractions) raise issues already addressed in other Larch interface languages.

In ML the equality test is defined for values constructed of integers, reals, strings, booleans, tuples, lists and datatypes. Equality testing is not possible for function types and abstract types. Therefore ML supports the notion of *equality* types, types that admit equality testing, which users may define through an **eqtype** declaration. Just as for a **type** declaration, Larch/ML allows a **based on** clause attached to an **eqtype** declaration.

ML's reference values denote store locations that can be updated. Since ML's basic type checking algorithm does not cope with assignments to polymorphic references, ML provides a class of type variables called *weak* type variables (*e.g.*, '1a), which offer a limited degree of polymorphism. Larch/ML allows weak type variables in function declarations and relies on the ML type checker to ensure that the rules applied are the same as those for checking a plain ML signature module.

In Larch/ML we treat ML's **datatype** declarations as "exposed" (concrete) types (à la Larch/C) since defining a **datatype** in a signature module is like exposing the representation of an abstract type to the module's client.

ML supports abstract data types through **abstraction** modules.[3] An abstraction, **abstraction** S: SIG = **struct** ... **end**, is a special kind of structure in ML that limits the view of the structure to be exactly what is specified in the signature SIG of the abstraction S. Thus Larch/ML interfaces might very well be implemented by abstractions. Rules for determining when an abstraction satisfies a Larch/ML interface would extend existing ML rules for determining when an abstraction satisfies an ML signature.

3 A New Application for LP

3.1 Specification Matching

We are exploring the idea of using the Larch Prover to do specification matching. In general we need to prove some formula that looks like:

$$s_i \textit{ satisfies } q$$

Consider the specification of the symbol table in Figure 5 to be the query q. We might want to retrieve different symbol table implementations that satisfy q. Suppose there are four implementations, each (informally) specified to have the following properties:

1. s_1 stores its entries in an order based on the lexicographic ordering of keys. No duplicates are stored.

2. s_2 stores its elements in insertion order. No duplicates are stored.

3. s_3 retains multiple, possibly duplicate, bindings to the same key, but upon lookup always returns the value most recently bound to the given key.

4. s_4 retains multiple, possibly duplicate, bindings to the same key, and upon lookup returns any one of the values bound to the given key.

Intuitively, the first three, and not the fourth, satisfy the query q and hence should be retrieved. (q maintains an unordered set of keys to which values are bound, and lookup always returns the most recently bound value.)

To make this intuition more concrete, let us first assume that the specification and query languages are drawn from the same logical language. Let this logical language be a first-order predicate language over equality, where terms are drawn from the term language of LSL, equality is defined as in LSL, and quantifiers are over state variables. We now need to show for a given module specification s_i:

$$s_i.Th \supseteq q.Th$$

[3] In our design of Larch/ML we choose to ignore **abstype**s since they have been supplanted by **abstraction**s.

where $s_i.Th$ stands for associated theory of the i^{th} specification and $q.Th$ stands for associated theory of the query q. It is beyond LP's ability to handle the above formula since it is a statement about theory containment.

However, now consider a subproblem–that of matching an individual function's specification–which would be necessary to partially prove the above. For retrieving ML functions whose specifications, s_i, are written in terms of pre- and post-conditions (as in Larch/ML), we need to match the corresponding pre- and post-condition predicates of a given function query q. A pre-condition is satisfied if the pre-condition of the query implies the pre-condition of the ML function. That is, the function's pre-condition can be weaker than the query's pre-condition, meaning that the function can be called in any context required by the query as well as other contexts. A post-condition is satisfied if the post-condition of the ML function implies the post-condition of the query. That is, the function's post-condition can be stronger than the query's post-condition, meaning that the function may produce results for any context required by the query as well as other contexts. We capture these ideas in terms of LP commands as follows:

> ... % other declarations and assertions
> **assert** S **generated by** emp, add
> ... % other declarations and assertions
> **prove** $q.pre \Rightarrow s_i.pre$ **by induction**
> **prove** $s_i.post \Rightarrow q.post$ **by induction**

where *x.pre* and *x.post* stand for the pre- and post-conditions of component *x*. Among the declarations and assertions in ..., we assert (as in the SymTab trait) that terms of sort S are generated by operator symbols emp and add. We then tell LP to prove the two implications by induction.

For example, suppose we want to retrieve the ML functions that assume an element *e* is in a collection *c* upon invocation and ensure that it no longer is upon return. That is,

$$q.pre \equiv isin(c, e), \text{ and}$$
$$q.post \equiv \sim isin(c\%, e).$$

Then looking at the delete function of Figure 5,

$$delete.pre \equiv isin(t, k), \text{ and}$$
$$delete.post \equiv t\% = rem(t, k).$$

To show that $isin(c, e) \Rightarrow isin(t, k)$ LP needs to do appropriate renaming of variables. More subtly, LP needs to show (or more realistically, to be told by the user through LSL's **includes** construct) that the theory of elements is included in the theory of keys and the theory of collections is included in the theory of symbol tables. To show $t\% = rem(t, k) \Rightarrow \sim isin(c\%, e)$ (*i.e.*, that an element removed from a collection implies that it no longer is a member of the collection when that collection is a table), LP needs to do equational reasoning. Figure 8 shows a proof sketch of what LP would need to prove given the appropriate human guidance.

Prove: $t\% = rem(t, k) \Rightarrow \sim isin(c\%, e)$.

Rename $c\%$ to be $t\%$ and e to be k by the same reasoning above about theory inclusion.

Then we need to show:

$$t\% = rem(t, k) \Rightarrow \sim isin(t\%, k).$$

Assume $t\% = rem(t, k)$. Show $\sim isin(t\%, k)$.
Hence, show $\sim isin(rem(t, k), k)$ by substitution for $t\%$.

Proof: By induction on t.
Base: $t = emp$
$\sim isin(rem(emp, k), k)$. Exempt case
since $rem(emp, k)$ is **exempt**.

Ind. step: Assume for all $t1, k$. $\sim isin(rem(t1, k), k)$. (IH)
Let $t = add(t1, k1, v1)$.
Show $\sim isin(rem(add(t1, k1, v1), k), k)$.
Case 1: $k = k1$. From the **then** part of equation for rem, we get:
$\sim isin(rem(t1, k), k)$ which is true by IH.
Case 2: $k \neq k1$. From the **else** part of equation for rem, we get:
$\sim isin(add(rem(t1, k), k1, v1), k)$
By the last equation in SymTab trait, we have:
$\sim isin(rem(t1, k), k)$ which is true by IH.

Figure 8: A Proof Sketch for Specification Matching

3.2 Open Problems

3.2.1 Re-engineering LP's interface

We begin with some obvious limitations of LP with respect to its user interface, rather than its functionality.

Translating a Larch/ML specification into LP input. As for other Larch interfaces, associated with a Larch/ML specification is a set of predicates. These predicates along with any of the trait theories the Larch/ML interface uses can be stored in a file in a format that LP interprets as commands. LP already has an **execute** command that has the effect of reading in a list of commands from a file.

User Interaction. Once the translation from a Larch/ML specification to LP input is done, then LP can essentially take over. Here is where the crux of the problem lies. LP is designed to be a proof checker, not an automatic theorem prover. Hence, the human user must painstakingly guide it into finding the proof of a formula. Though LP does invoke some built-in rules of inference automatically, the human user plays a critical role in seeing a proof to completion. Given the goal of retrieving software components from a program module library through specification matching, for typical queries, such interaction with LP would be an unacceptably high price to pay.

However, given a sufficiently restricted specification and query language, it is possible to do specification matching automatically. As a feasibility study, we recently experimented with using λProlog [7] to show the use of specifications as search keys [10]. Our implementation encodes both LSL traits and Larch interfaces into λProlog clauses. We used our λProlog version of Larch as a specification language to describe a toy library of ML functions. We relied on λProlog's built-in higher-order unifier to do the *satisfies* check "for free" (*i.e.*, no theorem-proving or user interaction is needed). In support of our hypothesis, we found that including more semantic information in the search key increases precision of a match. However, the primary limitation of our λProlog experiment is λProlog's inability to deal with equality; our specifications were similarly limited in expressibility.[4]

Vandevoorde uses LP as a backend to his Speckle checker [12]. He restricts his input to LP to that which LP can handle automatically. His work suggests that with similar kinds of restrictions, we can use LP as a backend for a restricted form of specification matching.

3.2.2 Extending LP's functionality

The problem of specification matching raises the following issues with respect to the current functionality of LP. These issues are orthogonal to the interface issues described above. and hence can be pursued in parallel. Extending LP's underlying logic would benefit not just our own research, but the Larch specification community and the theorem proving community in general.

Algorithmic. LP supports a limited form of universal quantification through its **deduction** rules. Every Larch interface language needs to be able to handle not just universal quantification in a more general form, but existential quantification as well. Other users of LP have also suggested adding existential quantification to support proofs of (bounded) liveness properties of concurrent systems.

For a Larch/ML interface, we need to be able to specify higher-order functions. If we were to extend the assertion language of an interface to support higher-order logic, then we would need to understand how to extend LP similarly. This extension would push against some known theoretical boundaries (*e.g.*, unification is known to be undecidable for orders greater than two and the result is unknown for order equal to two when variables are restricted to range over only types [5]); however, there exists a complete second-order matching algorithm [4]. Performing even first-order unification over equational theories raises some interesting issues. For example, EQLOG [2], which is based on Horn clause logic with equality rather than full first-order predicate calculus with equality, uses the technique of *narrowing* as a means of exploiting unification and term rewriting in one system. It would be interesting

[4]Sometimes a library component may satisfy a query, but without equality it cannot be proved. Thus, the query processor may miss some library components that would be of interest to the user.

to explore the design of efficient algorithms for performing (first-order) unification modulo "small" (application-specific) equational theories.

Finally, in order to perform the kind of specification matching suggested by the symbol table example, in principle we need to show that some theory includes another. A certain kind of theory containment (specifically, conservative extensions) was once considered in the design of LSL (*i.e.*, through the **imports** clause) but was recently discarded; one of the reasons was that LP cannot in general be used to check for conservative extension nor can reasonable sufficient conditions be found for any but the most trivial examples. It would be interesting to revisit this issue given a different motivating application.

Methodological. In reality, an LP proof of whether specification s_i satisfies a query q might best be broken into smaller subproofs. For example, a proof that a symbol table ML implementation satisfies the symbol table specification is necessarily going to depend on the proofs of each of the individual functions defined (*e.g.*, insert and delete). LP comes with limited facilities to help users manage proofs through a proof stack. A logically tree-structured proof is thus flattened into the LP-imposed linear structure of a stack. One possible extension to LP would be to let users maintain and traverse a proof tree, rather than a stack, to reflect more naturally proof structures and proof development.

LP's interface with the file system can be extended to support a library of proofs. The proof and specification subdirectories included with the sources to LP are a step in this direction. Organizing proof and specification libraries so they can be accessed from LP smoothly would be convenient for users.

4 Status of Design and Tools

We are in the midst of designing Larch/ML and have only been speculating on the use of LP for specification matching.

We are building a checker/translator for Larch/ML. Since a Larch/ML specification is an ML signature with specification clauses included as special comments, an ML compiler can check and compile a Larch/ML specification as a regular ML signature. The Larch/ML translator parses, checks, and translates the specification clauses. To check a Larch/ML specification fully, we first run an ML compiler over it and then run the Larch/ML translator over it.

Since we can use names from the ML signature in the specification clauses, the Larch/ML translator processes the ML signature, noting identifiers defined in the signature and their types. In processing the specification clauses, the translator does structural checks (*e.g.*, a **based on** clause follows a type declaration), namespace checks (*e.g.*, all identifiers are uniquely defined), and type checks (*e.g.*, all clauses are well-typed).

For a given specification, the Larch/ML translator generates a fully-checked abstract syntax tree that can be deposited into a persistent object base [9]. Then to search a software library, the Larch/ML translator would convert a specification query to an abstract syntax tree, which would be matched against all trees in the persistent object base. To match a query tree against a library component tree, we would first signature match on the trees. If this succeeds, we would then match each function in the query against a function in the component, and then specification match on their specifications using the Larch Prover as indicated in Section 3.

To date, we have completed structural and namespace checking for the Larch/ML translator and are implementing type checking. We have already added the capability of using a persistent object base to an ML compiler. We have implemented signature matching (on ML function types), but not yet specification matching.

Acknowledgments

We thank Greg Morrisett and Scott Nettles for their feedback on our design of Larch/ML and Stephen Garland for his help with using LP.

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597.

References

[1] Eric C. Cooper and J. Gregory Morrisett. Adding Threads to Standard ML. Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, December 1990.

[2] J. A. Goguen and J. Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Functional and Logic Programming*, pages 179–210. Prentice-Hall, 1986.

[3] J.V. Guttag and J.J. Horning. Introduction to LCL: A Larch/C Interface Language. Technical Report 74, DEC/SRC, July 1991.

[4] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[5] G.P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[6] K. Jones. Lm3: A Larch Interface Language for Modula-3. Technical Report 72, DEC/SRC, June 1991.

[7] D. A. Miller and G. Nadathur. Higher-order logic programming. In *Third International Conference on Logic Programming*, London, July 1986.

[8] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[9] Scott M. Nettles and J.M. Wing. Persistence + Undoability = Transactions. In *Proc. of HICSS-25*, January 1992.

[10] E.R. Rollins and J.M. Wing. Specifications as search keys for software libraries. In *Proceedings of the Eighth International Conference on Logic Programming*, Paris, June 1991.

[11] D. Sannella and A. Tarlecki. Program specification and development in Standard ML. In *Proceedings 12th ACM Symp. on Principles of Programming Languages*, pages 67–77, New Orleans, January 1985.

[12] M.T. Vandevoorde. Exploiting formal specifications to produce faster code. MIT Ph.D. dissertation in progress, 1993.

[13] J.M. Wing. *A Two-Tiered Approach to Specifying Programs*. PhD thesis, MIT, May 1983. Available as MIT-LCS-TR-299.

[14] Amy Moormann Zaremski. Using semantic information to find program modules. CMU Ph.D. dissertation in progress, 1993.