A Larch Specification of the Library Problem

Jeannette M. Wing

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

A claim made by many in the formal specification community is that forcing precision in the early stages of program development can greatly clarify the understanding of a client's problem requirements. We help justify this claim via an example by first walking through a Larch specification of Kemmerer's library problem and then discussing the questions that arose in our process of formalization. Following this process helped reveal mistakes, premature design decisions, ambiguities, and incompletenesses in the informal requirements. We also discuss how Larch's two-tiered specification method influenced our modifications to and extrapolations from the requirements.

1 Motivation and Contributions

A claim made by many in the formal specification community is that forcing precision in the early stages of program development can greatly clarify the understanding of a client's problem requirements. We help justify this claim by presenting a specification of Kemmerer's library database example.[1]

With this paper, we also contribute an example of a Larch two-tiered specification [7]. This contribution is significant because only a few formal specifications, let alone ones in Larch, have been published. In order to persuade people in the software community that formal specifications can be useful and need not be intimidating, it is important that more examples be given. Furthermore, we also use this opportunity to discuss how Larch's two-tiered specification method helps to guide the process of formalizing an informal problem statement.

In Section 2, we briefly review Larch; in Section 3, we present the informally stated requirements and our Larch specification of the library example; in Section 4, we discuss questions that arose in the process of writing our specification, the resulting modifications made in our response to the questions, and the influence our choice in specification methods had on our solution; in Section 5, we mention related work. We state some concluding remarks in Section 6.

[1]We emphasize that we do not mean to criticize Kemmerer's problem statement, but merely to use it as a vehicle with which to support the claim. This example appeared in Kemmerer's paper [11] and was listed as one of the problems of the *4th International Workshop on Software Specification and Design* [15].

2 Overview of Larch

What follows is enough of a review of Larch to understand the library example. We refer the reader to other publications (e.g., [7], [8], [17]) for more details.

A Larch specification has components written in two languages. A Larch *interface* language, e.g., Larch/CLU, is used to describe the observable behavior of program modules written in a particular programming language, e.g., CLU [12]. The Larch *Shared* Language is used to write *traits* that define the assertion language used in interface components. Essentially, interface specifications use predicates (pre- and post-conditions) to describe state transformations; traits use equational axioms to describe fundamental abstractions that are independent of state, and thus, of any programming language. Below, we illustrate the salient features of the Larch Shared Language and the Larch/CLU interface language through a simple example, which we revisit in the library example.

2.1 Larch Shared Language

Here is a trait useful for describing values for sets of elements.

```
Set: trait
  assumes Equality with [E for T]
  includes Integer
  introduces
    {}: → S
    add: S, E → S
    rem: S, E → S
     # ∈ #: E, S → Bool
    isEmpty: S → Bool
    |#|: S → Integer
     #∪#: S, S → S
  constrains [S] so that
   S generated by [{}, add]
   for all [s, s1: S, e, e1: E]
    rem({}, e) = {}
    rem(add(s, e), e1) =
         if e = e1 then rem(s, e1) else add(rem(s, e1), e)
    e ∈ {} = false
    e1 ∈ add(s, e) = (e = e1) ∨ e1 ∈ s
    isEmpty({}) = true
    isEmpty(add(s, e)) = false
    |{}| = 0
    |add(s, e)| = if e ∈ s then |s| else 1 + |s|
    {} ∪ s = s
    add(s, e) ∪ s1 = add((s ∪ s1), e)
    s ∪ s1 = s1 ∪ s
```

It contains a set of *operator* declarations, which follows the keyword **introduces**, and a set of equational axioms, which follows the **constrains** clause. An operator is declared by giving its name, e.g., add and rem, along with its *signature* (the *sorts*, e.g., S and E, of its domain and range). These signatures are used to sort-check *terms* (expressions) in much the same way as function calls are type-checked in programming languages. The set of equations following the **constrains** clause defines a meaning for the terms, more precisely, an equivalence relation on the terms, and hence, the values they denote. For example, from Set, we could prove that rem(add(add({}, 7), 7), 7) = {}.

A trait denotes a theory of typed first-order predicate calculus with equality. Each equation appearing in a trait is a formula in the trait's theory. A **generated by** clause adds an inductive rule of inference to a trait's theory. In the Set example, all values of sets of integers can be denoted by terms using only the operators, {} and add.

The Larch Shared Language also provides ways of putting traits together. A trait that **includes** another trait is textually expanded to contain all operator declarations, **constrains** clauses, **generated by** clauses, and axioms of the included trait. The meaning of the including trait is the meaning of the textually expanded trait. In the Set example, the signature and meaning of + comes from the Integer trait. Assumptions about operators appearing in a trait can be recorded in a trait **assumed** by another. For example, the equality symbol used between e and e1 in the second and fourth equations above, is assumed to satisfy the properties of an equivalence relation as specified in the Equality trait (see the Larch Library in [7]).

Renaming sort and operator identifiers is done through a **with** clause. In Set, we rename the sort T of Equality with E.

2.2 The Larch/CLU Interface Language

A Larch interface language is used to describe the behavior of program modules, in particular operations and abstract data types. Here is a Larch interface specification of a CLU operation that chooses and deletes an element from a set and returns the element:

```
choose = proc (a: set) returns (x: elem)
  requires ¬isEmpty(a)
  modifies at most [a]
  ensures a_post = rem(a, x) ∧ (x ∈ a)
```

The **requires** clause states a precondition that must hold when an operation is invoked. An omitted **requires** clause is interpreted as equivalent to **requires** true. In the above example, the caller of choose is required to pass a non-empty set argument; the implementor is allowed to rely on the pre-condition being met by the caller. A **modifies at most** clause identifies objects that the operation is allowed to change, e.g., choose shrinks its set argument. The **ensures** clause states a post-condition that the operation must establish upon termination. An unsubscripted argument formal, e.g., a, in a predicate stands for the value in the *pre* state. A return formal or a formal subscripted by *post*, e.g., a_{post}, stands for the value associated with the formal in the *post* state.

Since CLU allows for multiple termination conditions (normal and exceptional), Larch/CLU provides a way to specify exceptional termination. Instead of placing a pre-condition on the use of the choose operation, one would more typically specify it to return an exceptional condition:

```
choose = proc (a: set) returns (x: elem) signals (empty)
  modifies at most [a]
  ensures normally a_post = rem(a, x) ∧ (x ∈ a) except
             signals empty when isEmpty(a)
```

An interface specification of an abstract data type, T, consists of a set of interface specifications of the operations of T, a trait name, and a mapping between a sort name appearing in the trait and the type name T. The trait defines the meaning of the predicates in the body of an interface specification. For example, the meaning of rem, ∈, and = in the post-condition of choose comes from the Set trait. The sort-to-type name mapping defines what set of (sorted) terms denote the values of objects of the type. A typical header for an interface specification of the set type would be:

set **mutable type exports** create, insert, delete, size
  based on sort S **from** Set

where the **based on** clause indicates a mapping from S to set.

3 The Library Problem

3.1 Informal Requirements of the Problem

The problem as stated in the *ACM Software Engineering Notes* (K-SIG) [15] is as follows: "Consider a small library database with the following transactions:

1. Checkout a copy of a book. Return a copy of a book.
2. Add a copy of a book to the library. Remove a copy of a book from the library.
3. Get the list of books by a particular author or in a particular subject area.
4. Find out the list of books currently checked out by a particular borrower.
5. Find out what borrower last checked out a particular copy of a book.

There are two types of users: staff users and ordinary borrowers. Transactions 1, 2, 4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The database must also satisfy the following constraints:

1. All copies in the library must be available for checkout or be checked out.
2. No copy of the book may be both available and checked out at the same time.
3. A borrower may not have more than a predefined number of books checked out at one time."

A fourth constraint stated in Kemmerer's *IEEE Transactions of Software Engineering* paper (K-TSE) [11] was probably left out in the SIGSOFT version; since we refer to it later, we include it below:

4. A borrower may not have more than one copy of the same book checked out at one time.

3.2 A Formal Specification in Larch/CLU

We begin with a visualization of our model of the library depicted by the statechart [9] of Figure 1. There are two universes: people and books. Of the people, there are users and non-users of the library. Of users, there are regular and staff members. Of the books, there are library books and non-library books. Of the library books, there are those that are available for checking out and those that have been checked out. The set of users partition those books that are checked out. The diagram indicates whether a region within the library books denotes a set or multiset of elements. The traits of the following formal specification capture these relationships. The interfaces ensure that the required constraints are maintained as a book moves from one region to another, e.g., from being checked out to being available.

The presentation of our specification differs from its actual development. For pedagogical purposes, we walk through the pieces bottom-up, beginning with traits and then interfaces.

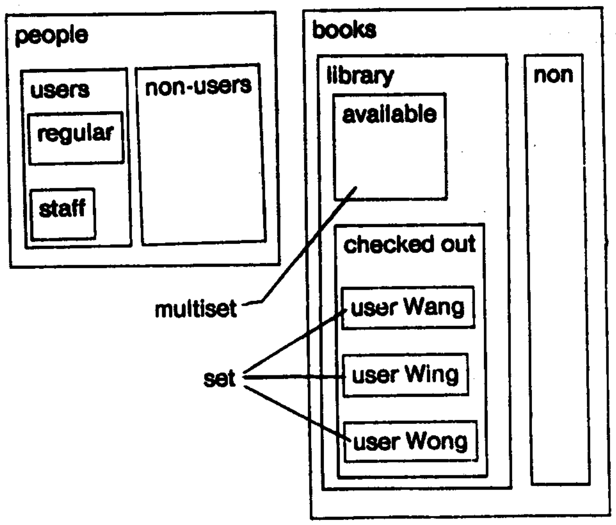


Figure 1: Statechart of People and Books

3.2.1 Traits

A book has a title, author, and subject:

Book: **trait**
  B **record of** [title: T, author: A, subject: S]

A user is regular or staff:

Status: **trait**
  St **enumeration of** [regular, staff]

Each user has a name, a status, and an associated set of books. We leave unspecified how a user's name is uniquely given, but assume that names satisfy the properties of an equivalence relation. A user is responsible for any book in his or her associated set of books.

```
User: trait
  assumes Equality with [N for T]
  includes Status, Book, Set with [BS for S, B for E]
  U record of [name: N, status: St, books: BS]
  introduces
     notStaff: U → Bool
     responsible: B, U → Bool
  constrains U so that for all [n: N, bs: BS, b: B, u: U]
     notStaff(<n, regular, bs>) = true
     notStaff(<n, staff, bs>) = false
     responsible(b, u) = b ∈ u.books
```

A library is a pair of a set of users and a multiset (bag) of available books. Thus, a library can have multiple copies of the same book, but users can have only a single copy of a book.

```
Library: trait
  includes User, Integer, Set with [US for S, U for E],
             Bag with [LBS for S, B for E],
  L record of [users: US, books: LBS]
  introduces
    allBooks: L → LBS
    usersBooks: US → LBS
    available: L, B → Bool
    checkedOut: L, B → Bool
    limit: L → Integer
  constrains L so that for all [l: L, u: U, us: US, b: B]
    allBooks(l) = l.books ∪ usersBooks(l.users)
    usersBooks({}) = {}
    usersBooks(add(us, u)) = if u ∈ us
             then setToBag(u.books) ∪ usersBooks(us)
             else usersBooks(us)
    available(l, b) = b ∈ l.books
    checkedOut(l, b) = b ∈ allBooks(l) ∧ b ∉ l.books
    limit(l) = aNumber % could tailor this to type of user
```

3.2.2 Interfaces

The previous set of traits is sufficient for us to write the interface specifications for the library transactions. The header for the library data type is:

```
library mutable type exports check_out, return, add,
        remove, titles_by_author, titles_by_subject,
        checked_out_what, last_responsible
 based on sort L from Library
    with [user for U, book for B, title for T,
                                author for A, subject for S]
```

where in the based on clause, we provide the sort-to-type name mappings, e.g., L to library, U to user, which identify the set of possible (sorted) values for each set of (typed) objects.

We now explain each of the operations of the library in turn. For a user to check out a book from the library, the book should be available, the user should not exceed the book limit, and the user should not have a copy of that book already. By checking out the book, the states of the library and the user may change. In the **normal** case, the user is responsible for another book and the library makes it unavailable for check out.

```
check_out = proc (l: library, u: user, b: book)
                        signals (notAvail, overLimit, hasCopy)
  modifies [l, u]
  ensures normally responsible(b, u_post) ∧
                        checkedOut(l_post, b) except
    signals notAvail when
                        b ∉ allBooks(l) ∨ checkedOut(l, b)
    signals overLimit when size(u.books) = limit(l)
    signals hasCopy when responsible(b, u)
```

A user can return a book to the library if he or she was responsible for it; if so, the book is now available for check out and the user is no longer responsible for it.

```
return = proc (l: library, u: user, b: book)
                                signals (notResponsible)
  modifies [l, u]
  ensures normally available(l_post, b) ∧
                  ¬responsible(b, u_post) except
    signals notResponsible when ¬responsible(b, u)
```

A user can add a book to the library if the user is a staff person. No check is made to see if the book is in the library already since multiple copies of the same book are allowed.

```
add = proc (l: library, u: user, b: book)
                                signals (notAuthorized)
  modifies [l]
  ensures normally available(l_post, b) except
    signals notAuthorized when notStaff(u)
```

A user can remove a book from the library if the user is a staff person and if the book is not checked out by any user, i.e., available.

```
remove = proc (l: library, u: user, b: book)
                    signals (notAuthorized, notAvailable)
  modifies [l]
  ensures normally b ∉ allBooks(l_post) except
    signals notAuthorized when notStaff(u)
    signals notAvailable when ¬available(l, b)
```

The set of titles of library books returned may be by author or subject.

```
titles_by_author = proc (l: library, a: author)
                                        returns (ts: set[title])
  modifies nothing
  ensures ∀ b: B [(b ∈ allBooks(l) ∧ b.author = a) ⇒
                                        b.title ∈ ts]

titles_by_subject = proc (l: library, s: subject)
                                        returns (ts: set[title])
  modifies nothing
  ensures ∀ b: B [(b ∈ allBooks(l) ∧ b.subject = s) ⇒
                                        b.title ∈ ts]
```

The set of titles of books currently checked out by some user, who, is returned only if the asker is a staff person or is the same as who.

```
checked_out_what = proc (who, asker: user)
           returns (ts: set[title]) signals (notAuthorized)
  modifies nothing
  ensures
    normally ∀ b: B [b ∈ who.books ⇒ b.title ∈ ts]
    except
      signals notAuthorized when
                   ¬[asker.status = staff ∨ who = asker]
```

The user last responsible for a book is returned only if the asker is a staff person, the book is a library book, and the book is currently checked out.

```
last_responsible = proc (l: library, asker: user, b: book)
                                        returns (u: user)
      signals (notAuthorized, notABook, notCheckedOut)
  modifies nothing
  ensures normally responsible(b, u) except
    signals notAuthorized when notStaff(asker)
    signals notABook when b ∉ allBooks(l)
    signals notCheckedOut when available(l, b)
```

From the traits and the interface specifications together, one can show that the requirements of the problem statement are satisfied. Other properties are also provable. For example, the following three theorems are provable from the Library trait.

1. If a book is not checked out and is not available, then it is not a library book at all:

$[\neg checkedOut(l, b) \wedge \neg available(l, b)] \Rightarrow b \notin allBooks(l)$

2. A library book is available if and only if it is not checked out:

$b \in allBooks(l) \Rightarrow [available(l, b) \Leftrightarrow \neg checkedOut(l, b)]$

3. If a book is checked out, then some user must have it:

$checkedOut(l, b) \Rightarrow \exists u: U\ [u \in l.users \wedge responsible(b, u)]$

4 Discussion

In the course of formalizing the library example, a number of questions arose whose answers helped shape our solution. Some questions are attributable to the informality of the problem requirements and some to the particular formal specification method chosen.

4.1 Modifications to the Requirements

Typical problems with informal requirements appeared in the library example; we modified the requirements accordingly.

Unintentional mistake: That Transaction 1 be restricted to only staff users is probably an unintentional mistake in the problem requirements. Checking out and returning books should be made available to ordinary borrowers as well as library staff. This modification to the requirements is consistent with Kemmerer's statement of K-TSE.

Overspecification: Returning sets of books is less restrictive than returning lists of books. Sets allow for the possibility of non-linearly formatted, and possibly unordered output, e.g., tables, charts, or pictures. Multisets would allow for multiple copies to be returned as well.

Ambiguity: What is the distinction between a book and a copy of a book? Without Kemmerer's fourth functional requirement of K-TSE, there seems to be no reason for the distinction. For example, the second requirement says "No copy of the book may be both available and checked out at the same time." Surely no book, let alone a copy of one, may be both available and checked out at the same time.

The fourth requirement makes the problem more interesti g, so the Larch specification allows the library to maintain multiple copies of a book, but any one user to have only one copy. Here, "copy" is taken to mean a book object with the same title, author, and subject (akin to equality on a record type). Hence, we model the books in a library as a bag (to allow for multiple copies) and the books associated with individual users as a set (to disallow for multiple copies). Notice that two different users may have different copies of the same book and the library may still have available for check out other copies of a checked out book.

Inconsistencies or *contradictions* are another common problem with informally stated requirements. None were detected in K-SIG.

4.2 Extrapolations from the Requirements

Incompletenesses of the requirements led to the following extrapolations:

Exceptional conditions: Many possible "error" cases that could reasonably arise in a realistic library are not addressed explicitly by the requirements. Larch/CLU provides a useful mechanism to demarcate and handle exceptional cases; thus the Larch/CLU interface handles some of them to make the problem more realistic. For example, if a book is not checked out, the last_responsible procedure will terminate signalling notCheckedOut to indicate that the book is still available. The specification still does not handle all cases, e.g., none of the library operations specify behavior for when a user argument is not a member of the library's set of users.

Initialization: A create operation for the library data type is an obvious omission. To make the specification more complete, we would add the following operation:

```
create_library = proc () returns (l: library)
  ensures normally l = <{}, {}> ∧ new [l]
```

Adequacy: The set of library operations does not include ways to add and remove users. For example, for a staff person to add a user (by name), we would add the following two operations:

```
add_regular = proc (l: library, asker: user, n: name)
                          signals (notAuthorized, alreadyIsUser)
  modifies [l.users]
  ensures
    normally l.users_post = add(l.users, <n, regular, {}>)
    except
      signals notAuthorized when notStaff(asker)
      signals alreadyIsUser when
                          ∃ u':user [u'.name = n ∧ u' ∈ l.users]

add_staff = proc (l: library, asker: user, n: name)
                          signals (notAuthorized, alreadyIsUser)
  modifies [l.users]
  ensures
    normally l.users_post = add(l.users, <n, staff, {}>)
    except
      signals notAuthorized when notStaff(asker)
      signals alreadyIsUser when
                          ∃ u':user [u'.name = n ∧ u' ∈ l.users]
```

Notice that the assumption that names are unique is used to determine whether or not a user is a member of the set of library users.

For removing a user, we would add:

```
remove_user = proc (l: library, asker: user, u: user)
                          signals (notAuthorized, hasBooks)
  modifies [l.users]
  ensures normally l.users_post = rem(l.users, u) except
    signals notAuthorized when notStaff(asker)
    signals hasBooks when ¬isEmpty(u.books)
```

4.3 Influence of Specification Method

Larch's two-tiered specification method encourages specifiers to identify appropriate specification modules and abstractions when attacking a given problem. Larch/CLU also encourages specifiers to consider explicitly what exceptional situations may arise and how to treat them.

Traits are pieces of specifications that use other specifications (witness the use of the Set, Bag, and Record traits in the Library trait). Larch provides many ways to put traits together, thus encouraging a style of writing small specifications and reusing them for different purposes. The Larch library specification illustrates two of the more common reuses of traits: inclusion of traits into another and renaming identifiers of one trait by another. Furthermore, each interface specification uses traits and is itself a modular unit of specification.

Writing Larch interface specifications forces one to focus on choosing appropriate data abstractions for a system, thus encouraging a modular decomposition based on what data is being manipulated. For the library data type, we needed to answer "What are the properties of a library that we want to maintain?" and "What is an appropriate set of operations for a library?" Answering the first question helped determine what to write in the traits. Answering the second helped reveal the incompletenesses mentioned in the previous section as well as raise the issue that perhaps some operations, e.g., what_checked_out, more properly belong outside the type.

Larch's two-tiered method allows the specifier to separate out the functions to be implemented, i.e., the operations, and functions used to help define program behavior, i.e., operators in traits. Trait operators can be viewed as hidden or auxiliary functions not intended to be implemented. A consequence of this separation is that operators are often introduced into traits to make interface predicates simpler and often more concise. For example, the "notStaff" operator of the User trait and the "responsible" operator of the Library trait are strictly not needed since they can be expressed using previously defined operators. In writing the interfaces for the library operations, however, instead of writing "¬(u.status = staff)," we found it more convenient to write and easier to read the predicate "notStaff(u)," which appears in six of the library operations (including the three for adding and removing users).

Finally, there is a subtlety in the Larch/CLU specification that reflects the subtle difference between an object and its value in CLU. A *typed* object has a value denoted by a *sorted* term. CLU objects are defined by interface specifications of data types; their values are defined by traits. Thus, no trait is introduced to define values for copies of books, only values for books. We introduced a book type (whose specification is omitted here for brevity) so that distinct book objects with the same value (as defined in the trait Book) are copies of one another.[2] We could have removed this subtlety at the trait level by defining a way to uniquely identify each book copy (version number, UPC code, etc.) and including that unique identifier as part of the value of the book (in addition to author, subject, and title).

5 Related Work

Since Kemmerer uses Ina Jo of the Formal Development Methodology (FDM)[3] to specify the library example, we begin by comparing Ina Jo and Larch, and then follow with a comparison of Larch and other widely-known specification languages.

5.1 Comparison to Ina Jo

Ina Jo and Larch have some similarities. Constant, variable, and function variables in Ina Jo are analogous to Larch trait operators. Axioms in Ina Jo are analogous to trait equations. Transforms are analogous to interfaces. The tradeoff between refcond/effect and requires/ensures is similar: an effort to place no pre-condition on any of the procedures is made in both specification methods.

Ina Jo has specific clauses for stating initial conditions, invariants over states, and invariants over pairs of states. Larch has no explicit means for stating initial conditions nor for stating invariants over pairs of states. Invariants over states must be either written in traits or proven from Larch trait and interface specifications. Thus, whereas an Ina Jo specifier will naturally consider the initial state, a Larch specifier may overlook this case; whereas an Ina Jo specifier will naturally record state invariants, a Larch specifier may neglect to include them explicitly in the equations or consequences of a trait.

The three most notable differences between Larch and Ina Jo are in the treatment of modularity, abstraction, and exceptions. In Ina Jo, one writes a single top-level specification that describes *in toto* the behavior of an entire system. Within such a specification, one focuses more on state transitions and not on the semantics of the objects that make up a state. It provides a means to introduce types, but they remain uninterpreted unless axioms are given to describe the types' values. Finally, no mechanism exists for exceptions in Ina Jo.

5.2 Other Languages

Some formal specification languages that are similar to Larch are CLEAR [4], ACT-ONE [5], and SPECIAL [14]. Other specification languages, e.g., Iota [13], Z [1, 16], VDM's Meta-IV [3], and Gypsy [6] are based on different, often richer, semantic models.

One important difference between CLEAR and Larch is that specifications written in CLEAR have no simple way of specifying side effects and error handling of procedures. We use a Larch interface language to deal with issues like side effects and errors. One difference between the two languages, CLEAR and ACT-ONE, and Larch is that their semantics are described in terms of models, e.g., initial algebras, whereas ours are described in terms of theories, e.g., sets of first-order formulae. Unlike in Larch, none of CLEAR, Iota, and ACT-ONE attempts to separate specifying programming language issues like side effects, modularization, and parameterization from specifying fundamental abstractions.

SPECIAL separates an "assertion" part, analogous to our Shared Language component, from a "specification" part, analogous to our interface language component. A major difference between SPECIAL and our work is that in SPECIAL, types used in the specification part are defined in the assertion part, whereas we define types in interface language components ("specification" parts). Also, in SPECIAL a type is restricted to be either a primitive type, a subtype, or a structured type, each of which comes with a

[2]CLU provides *equal* and *similar* operations on objects that are used to check whether two objects are identical or whether just their values are identical.

[3]Ina Jo and FDM are trademarks of System Development Corporation, formerly a Burroughs Company, now UNISYS.

set of pre-defined functions. Larch does not restrict the assertion language to be based on a fixed set of primitives, and allows the specifier to use the Shared Language component to define exactly the assertion language desired. Since the assertion language in SPECIAL is restricted, most of the work of writing a specification is done in the specification part. We take the opposite viewpoint and expect most of the work of writing a specification to be done in the Shared Language component ("assertion" part).

6 Concluding Remarks

Larch is ideally suited to specify problems such as the library or formatter problems of [15]. It never was originally intended to be used to specify concurrent systems, and thus would not be suited to specify the liveness properties of the lift problem. Since it focuses more on data than control, specifying the control requirements of either the lift or heating system problems would be less straightforward than specifying the data invariants of the library and formatter problems. It is easier in Larch to assert what properties of data must hold than to assert that a sequence (or interleaving) of state transitions must (or must not) occur.

More recently, however, people have applied Larch to the domain of concurrency. The language as it is currently defined is used as a formal specification language for characterizing execution sequences of concurrent histories that are *linearizable* [10]. Others have extended the language with **when** conditions to handle the explicit specification of synchronization conditions [2]. In both cases, the two-tiered method has been natural to apply and useful for separating between different levels of abstraction.

References

[1] J.R. Abrial. *The Specification Language Z: Syntax and Semantics*. Technical Report, Programming Research Group, Oxford University, 1980.

[2] A. Birrell, J.V. Guttag, J.J. Horning, and R. Levin. The Threads Synchronization Primitives. 1986.private communication.

[3] D. Bjorner and C.G. Jones (Eds.). *Lecture Notes in Computer Science*. Volume 61: *The Vienna Development Method: the Meta-language*. Springer-Verlag, Berlin-Heidelberg-New York, 1978.

[4] R.M. Burstall and J.A. Goguen. An Informal Introduction to Specifications Using CLEAR. In Boyer and Moore (editors), *The Correctness Problem in Computer Science*. Academic Press, 1981.

[5] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*. Springer-Verlag, Berlin, 1985.

[6] D.I. Good, R.M. Cohen, C.G. Hoch, L.W. Hunter, and D.F. Hare. *Report on the Language Gypsy, Version 2.0*. Technical Report ICSCA-CMP-10, Certifiable Minicomputer Project, The University of Texas at Austin, September, 1978.

[7] J.V. Guttag, J.J. Horning, and J.M. Wing. *Larch in Five Easy Pieces*. Technical Report 5, DEC Systems Research Center, July, 1985.

[8] J.V. Guttag, J.J. Horning, and J.M. Wing. The Larch Family of Specification Languages. *IEEE Software* 2(5):24-36, September, 1985.

[9] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 1987. to appear.

[10] M.P. Herlihy and J.M. Wing. Axioms for Concurrent Objects. In *Proc. Fourteenth ACM Symp. on Principles of Programming Languages*. Munich, W. Germany, January, 1987.

[11] R.A. Kemmerer. Testing Formal Specifications to Detect Design Errors. *IEEE Transactions on Software Engineering* SE-11(1):32-43, January, 1985.

[12] B.H. Liskov, et al. *Lecture Notes in Computer Science*. Volume 114: *CLU Reference Manual*. Springer-Verlag, 1981.

[13] R. Nakajima, M. Honda, and H. Nakahara, H. Hierarchical Program Specification and Verification-- A Many-sorted Logical Approach. *Acta Informatica* 14:135-155, 1980.

[14] L. Robinson and O. Roubine. *SPECIAL - A Specification and Assertion Language*. Technical Report CSL-46, Stanford Research Institute, Menlo Park, Ca., January, 1977.

[15] Call for Papers. Problem Set for the 4th International Workshop on Software Specification and Design. *ACM Software Engineering Notes* , April, 1986.

[16] B. Sufrin, C. Morgan, I. Sorensen, and I. Hayes. *Notes for a Z Handbook: Part I--The Mathematical Language*. Technical Report, Programming Research Group, Oxford University Computing Laboratory, August, 1984.

[17] J.M. Wing. *A Two-Tiered Approach to Specifying Programs*. Technical Report MIT-LCS-TR-299, MIT Laboratory for Computer Science, Cambridge, Mass., June, 1983.

[illegible]

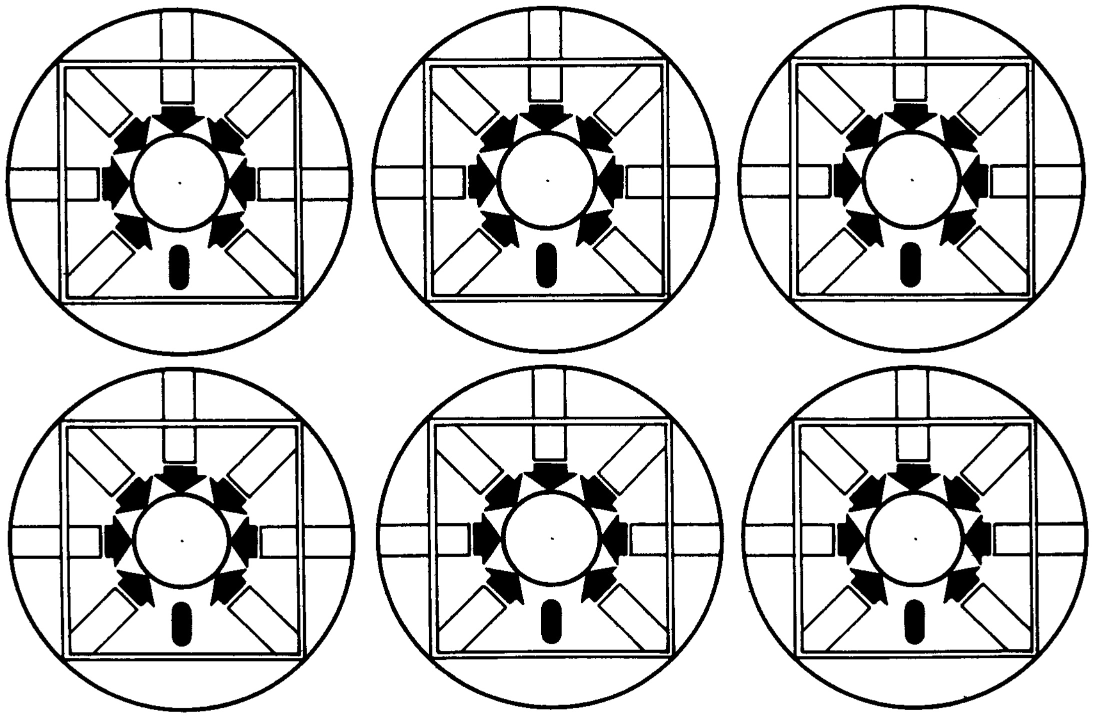
Proceedings

FOURTH INTERNATIONAL WORKSHOP ON SOFTWARE SPECIFICATION AND DESIGN

APRIL 3-4, 1987
MONTEREY, CALIFORNIA, USA

Sponsored by:

IEEE Computer Society
LCRST, Japan
Alvey Directorate, UK
ACM SIGSOFT
Agence de l'Informatique
& AFCET, France



IEEE Computer Society Order Number 769
Library of Congress Number 86-83318
IEEE Catalog Number TH0181-8
ISBN 0-8186-0769-6
SAN 264-620X

THE COMPUTER SOCIETY OF THE IEEE
THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.
IEEE
COMPUTER SOCIETY PRESS