Writing Larch Interface Language Specifications

JEANNETTE M. WING
Carnegie-Mellon University

Current research in specifications is emphasizing the practical use of formal specifications in program design. One way to encourage their use in practice is to provide specification languages that are accessible to both designers and programmers. With this goal in mind, the Larch family of formal specification languages has evolved to support a two-tiered approach to writing specifications. This approach separates the specification of state transformations and programming language dependencies from the specification of underlying abstractions. Thus, each member of the Larch family has a subset derived from a programming language and another subset independent of any programming languages. We call the former interface languages, and the latter the Larch Shared Language.

This paper focuses on Larch interface language specifications. Through examples, we illustrate some salient features of Larch/CLU, a Larch interface language for the programming language CLU. We give an example of writing an interface specification following the two-tiered approach and discuss in detail issues involved in writing interface specifications and their interaction with their Shared Language components.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages*; *methodologies*; D.2.2 [**Software Engineering**]: Tools and Techniques—*modules and interfaces*; D.3.2 [**Programming Languages**]: Language Classifications; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types*; *modules, packages*; *procedures, functions and subroutines*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions*; *specification techniques*

General Terms: Design, Languages, Verification

Additional Key Words and Phrases: CLU, Larch Shared Language, two-tiered specification approach

1. INTRODUCTION

1.1 Motivation

Current research in specifications is emphasizing the practical use of formal specifications in the programming process [19, 20]. People have already benefited from using informal specifications in most phases of this process. Writing informal specifications is widely accepted as a useful way of organizing ideas,

This research was supported in part by the National Science Foundation under grant ECS-8403905 at the University of Southern California, and is currently supported in part by the NSF under grant DMC-85 19254 at Carnegie-Mellon University. In addition to support in part by the NSF to the author, the Larch Project has been supported at the Massachusetts Institute of Technology's Laboratory for Computer Science by the NSF under grant MCS-8119846, and by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N0014-83-K-0125, by the Digital Equipment Corporation at its Systems Research Center, and by the Xerox Corporation, Palo Alto Research Center.
Author's address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213-3890.

documenting design decisions, and informally arguing the correctness of programs. Software design methods that include some form of informal specification have been in use in industry for some time [9, 30, 34, 53].

Thus far, formal specifications have played a less influential role in the programming process than have informal specifications. We believe that using formal specifications early in the process (i.e., in the design phase) can be especially beneficial. A specification is formal if it is written in a language with explicitly and precisely defined syntax and semantics. Hence, one virtue of formal specifications is their precision. Precision leaves no room for ambiguity. The process of writing formal specifications can often reveal ambiguities in a client's problem statement and errors in a program's design. Uncovering bugs early can thus save the cost of uncovering them later during testing and debugging. Precision also implies that we can formally argue the correctness of programs. Another virtue of formal specifications is their amenability to machine manipulation. With help from appropriate machine support (e.g., theorem provers), we can handle more specifications, and more complex ones, and thus formally reason about a larger set of specifications and programs than if we had to rely on pencil and paper only.

In this paper we focus on the formal specifications of program modules. We are interested in specifying program modules as a means of specifying a program composed of them. Given a specification of a program module, a program designer can choose to use the module without knowing how it is to be implemented. Similarly, a programmer can implement the module without knowing how it is to be used. Thus, from either the designer's or implementer's point of view, replacing one correct implementation of the module by another should not affect the program's design.

1.2 Context: Larch's Two-Tiered Approach

The Larch Project, ongoing at the MIT Laboratory for Computer Science and the DEC Systems Research Center, is developing tools, languages, and techniques intended to aid in the productive use of formal specifications. The set of tools includes language-sensitive editors and semantic checkers based on a powerful theorem prover [13, 36]. The Larch family of specification languages [22] includes the design and formal definitions of the Larch Shared Language [24] and Larch interface languages [51]. These languages were designed to support the specification technique called "the two-tiered approach," which was first introduced in [19] and elaborated upon in the author's Ph.D. thesis [51].[1]

The two-tiered approach separates the specification of underlying abstractions from the specification of state transformations. The specification of each program module has a component on each tier. The *Larch Shared Language* is used for the component that specifies underlying abstractions, and a *Larch interface language* is used for the component that specifies state transformations.

[1] We refer the reader to [23] for a compendium of five Larch papers bound in one document. Each is a minor variation of this paper, [22, 24] or [25].

We gain the following three advantages in separating a specification into two tiers:

—A separation of concerns.
—A division of effort. Shared Language components can be written independently of interface language components. Thus the task of specifying a program can be divided according to the skills of the specifiers on a programming project.
—Reusability. Shared Language components can be reused by different interface language components. Some of them can be developed for particular applications; a few central ones can be useful in many applications.

We believe that for specifications of program modules, the environment in which a module is embedded, and hence the nature of its observable behavior, is likely to depend in fundamental ways on the semantic primitives of the programming language. Attempts to hide this dependence will make specifications more obscure to both the module's users and its implementers. Thus, we intentionally make an interface language dependent on a target programming language, and keep the Shared Language independent of any programming language. To capitalize on our separation of a specification into two tiers, we isolate programming language dependent issues—such as side effects, error handling, and resource allocation—into the interface language component of a specification.

We use the term "interface" because an interface specification defines only the observable behavior of a program module. Users of a module read its interface specification to understand its behavior, without considering its internal structure. We use the term "shared" because all the Larch interface languages rely on the same language to define underlying abstractions.

1.3 Focus of this Paper

This paper focuses on Larch interface language specifications, whereas previous papers on the Larch Project have either focused on only Larch Shared Language specifications [21, 24, 25], or given only an introductory overview to both [22, 28]. The purpose of this paper is to explain in more detail what interface language specifications are, what they look like, and how they are intended to be written, used, and evaluated. A significant subgoal is to explain their interaction with Shared Language specifications. More specifically, in Section 2 we present an informal description of Larch/CLU, an interface language for the programming language CLU [38, 40]; in Section 3, we illustrate how one might write a two-tiered specification incrementally by following the two-tiered approach; in Section 4, we discuss some consequences of the two-tiered approach: how the two tiers interact and how an interface language supports our approach. In Section 5, we discuss further work, and include some suggestions for designing one's own interface language.

1.4 Related Work

1.4.1 *Specification Approaches.* Formal specifications have been used extensively to describe simple programs and abstract data types, leading to two different approaches, sometimes referred to as "operational" and "definitional." A survey of these approaches can be found in [37] and [39]. In the operational

approach, one gives a method of constructing the program or abstract data type. Examples of the operational approach include Parnas's work on state-machines [44], Robinson and Roubines's extensions to them with V-, O-, and OV-functions [45], Berzins's abstract models [4], and Jones's model-oriented specifications [31].

In the definitional approach of specifying a program or an abstract data type, one gives a list of its desired properties, not a method of constructing it. The definitional approach can be broken into two categories, sometimes referred to as "axiomatic" and "algebraic." The axiomatic approach stems from Hoare's work on proofs of correctness of programs [26] and of implementations of data types [27], where predicate logic pre- and post-conditions are used for the specification of the input–output behavior of programs and of each operation of an abstract data type. Other work using the axiomatic approach is described in [42] and [47]. The algebraic approach uses axioms to specify properties of programs and abstract data types, but the axioms are restricted to equations. This approach defines data types to be heterogeneous algebras [5]. Much work has been done on the algebraic specification of abstract data types [2, 7, 10, 17, 32, 50, 56], including the handling of error values [2, 14, 33], nondeterminism [33], and parameterization [11, 15, 49].

Our work is related to both the axiomatic and algebraic approaches. In the interface language component of a two-tiered specification, we use an axiomatic approach. In the Shared Language component, we use an algebraic approach.

1.4.2 *Specification Languages.* Some of the more widely known specification languages are CLEAR [8], Iota [43], ACT-ONE [12], SPECIAL [45], Z [1, 48], VDM's Meta-IV [6], Ina Jo [46], Gypsy [16], and PAISLey [55]. Of these, the ones most closely related to ours are CLEAR, Iota, ACT-ONE, and SPECIAL.

CLEAR, Iota, and ACT-ONE support the definitional approach of describing abstract data types. One important difference between the two languages, CLEAR and Iota, and ours is that specifications written in CLEAR or Iota have no simple way of specifying side effects and error handling of procedures. We use the interface language component of a two-tiered specification to deal with issues like side effects and errors. One difference between the two languages, CLEAR and ACT-ONE, and ours 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 formulas). 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, Z, Meta-IV, Ina Jo, Gypsy, and PAISLey support the operational approach; of these six, only SPECIAL is closely related to our two-tiered viewpoint. SPECIAL separates an "assertion" part, analogous to our Shared Language component, from a "specification" part, analogous to our interface language component. A major difference beween 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 a primitive type, a subtype, or a structured type, each of which comes with a set of predefined 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).

2. AN INFORMAL LOOK AT THE LARCH/CLU INTERFACE LANGUAGE

In this section, we intend to give the reader the flavor of an interface language, and in particular, one for the programming language CLU. Henceforth we use "Larch/CLU" for "the Larch/CLU Interface Language." Instead of presenting a formal description of Larch/CLU, we illustrate its salient features through some simple examples. Its complete formal definition can be found in [51].

Since the meaning of a Larch interface language is dependent on both the Larch Shared Language and a programming language, before we can describe Larch/CLU, we need to describe the Larch Shared Language and CLU. In Section 2.1 we present only those details of the Larch Shared Language that are essential to understand the examples of interface specifications we present in this paper. In Section 2.2 we do the same for CLU.[2] We refer the interested reader to [24] for further details of the Larch Shared Language; to [40] for details about CLU. In Section 2.3 we give an example of a Larch/CLU *procedure specification*, and in Section 2.4 we give an example of a Larch/CLU *cluster specification*. In Section 2.5 we briefly summarize the interface language features introduced through the examples.

2.1 An Overview of the Larch Shared Language

The unit of encapsulation in the Larch Shared Language is called a *trait*. Figure 1 shows a trait useful for describing values for sets of integers. The example is similar to a conventional algebraic specification in the style of [18] and [41].

A trait contains a set of *operator* declarations, which follows the keyword **introduces**, and a set of equational axioms, which follows a **constrains** clause. An operator is declared by giving its name along with its *signature* (the *sorts* 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. Two things that distinguish traits from specifications written in typical algebraic specification languages are (1) the name of a trait (e.g., SetOfE) is different from all *sort* and *operator* identifiers (e.g., SI, E, remove, card), appearing within a trait. A trait need not correspond to a single abstract data type, and often does not; (2) the **constrains** list contains all the operators that the immediately following axioms are intended to constrain. In the SetOfE trait, the **constrains** list informs us that the axioms are not to put any constraints on the properties of **if then else**, false, 0, +, and =, despite their occurrence in the axioms.

[2] In particular, in this paper we ignore the following features of CLU: iterators, own data, and parameterized modules. They are all carefully treated in [51].

```
SetOfE: trait
  includes Integer
  introduces
    empty: → SI
    add: SI, E → SI
    remove: SI, E → SI
    has: SI, E → Bool
    isEmpty: SI → Bool
    card: SI → Int
  constrains empty, add, remove, has, isEmpty, card so that
    SI generated by [empty, add]
    for all [s: SI, e, e1: E]
      remove(empty, e) = empty
      remove(add(s, e), e1) =
        if e = e1 then remove(s, e1) else add(remove(s, e1), e)
      has(empty, e) = false
      has(add(s, e), e1) = if e= e1 then true else has(s, e1)
      isEmpty(empty) = true
      isEmpty(add(s, e)) = false
      card(empty) = 0
      card(add(s, e)) = if has(s, e) then card(s) else 1 + card(s)
```

Fig. 1. SetOFE trait specification.

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. Saying that sort S is **generated by** a set of operators, Ops, asserts that each term of sort S is equal to a term whose outermost operator is in Ops. In the SetOfE example, all values of sets of integers can be denoted by terms using only the operators, empty and add. Together, the **introduces**, **constrains**, and **generated by** clauses, the "inequation" ¬(true = false), and propositional and quantified tautologies define the first-order theory of a trait.

The Larch Shared Language also provides ways of putting traits together, one of which is through an **includes** clause. 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 SetOfE example, the signature and meaning of + comes from the Integer trait.

2.2 An Overview of CLU

CLU has the primitive notions of *object* and *state*. An object is an entity that can be manipulated by a program. Two important properties of an object are its *type*, which never changes, and its *value*, which may change. A state consists of a set of objects, a mapping from program variables (object identifiers) to objects, and a mapping from objects to values. Two important observable state changes are when a new object is created and when the value of an existing object changes. An object whose value can change is said to be *mutable*; one whose value cannot change is said to be *immutable*. A type is mutable if objects of that type are mutable. For example, integers are immutable, but arrays are mutable in CLU.

```
choose = proc (s: set) returns (i: int)
  uses SetOfE with [set for SI, int for E]
    requires ¬ isEmpty(s_pre)
    modifies at most [s]
    ensures has(s_pre, i_post) ∧ (s_post = remove(s_pre, i_post))
  end
```

Fig. 2. A choose procedure specification.

In CLU, an object *A* can be the value of another object *B*, in which case we say "*B* contains *A*." Sharing of objects arises when two or more objects contain the same object. Because of sharing of mutable objects, it is not sufficient that the value of a containing object refer to the value of the contained object; it must refer to the contained object itself, that is, its identity. Therefore, we must be able to distinguish in our specifications between an object's identity and its value.

A CLU program consists of a set of modules, each of which is either a *procedure* or a *cluster*. A procedure performs an action on a set of objects and terminates returning a set of objects. Communication between a procedure and its invoker occurs through these objects. A cluster names a type and defines a set of procedures that create and manipulate objects of that type. Users of this type are constrained to treat objects of the type abstractly. That is, objects can be manipulated only via the procedures defined by the cluster, so, in particular, information about how objects are represented may not be used.

It is important not to confuse an object and its type, which are CLU concepts, with a term and its sort, which are Larch Shared Language concepts. The connection between the CLU and the Larch Shared Language concepts is that (typed) objects have values that are denotable by (sorted) terms. Through the Larch/CLU interface specifications of procedures and clusters, we establish a link between the values that objects can have and the terms defined by Shared Language components.

2.3 Larch/CLU Procedure Specifications

Figure 2 gives a Larch/CLU specification of a choose procedure that selects a member of a set, removes it, and returns it. It consists of a *header*, a *link* to its Shared Language component, and a *body*. The header indicates that the input argument is of type set, and the output result is of type int. The identifiers, s and i, denote objects, not values. The link from the interface component to the Shared Language component is established by the **uses** clause, which names the *used trait*, SetOfE, and provides a mapping from type to sort identifiers. The body contains a **requires/modifies/ensures** triple. The precondition in the **requires** clause of choose is an assertion that is satisfied if the initial value of the input argument is not empty. The **modifies at most** clause asserts that the choose procedure may mutate no object other than the object bound to s. The postcondition in the **ensures** clause is an assertion about the initial and final values of the set object and the final value of the int object. The operator names, isEmpty, has, and remove, and the meaning of the equality symbol, =, all come from SetOfE.

Associated with a procedure specification is the predicate over two states,

$$\text{Pre} \Rightarrow (\text{Modifies} \wedge \text{Post})$$

where Pre and Post are the assertions in the **requires** and **ensures** clauses, respectively, and Modifies is the assertion associated with the **modifies at most** clause. The clause **modifies at most** [$x1, \ldots, xn$] asserts that the procedure changes the value of no object in the environment of the caller except possibly some subset of $\{x1, \ldots, xn\}$.

It is important to notice the following points about a procedure specification:

(1) We distinguish between an object and its value by using a plain object identifier (e.g., s) to denote an object, and a subscripted object identifier (e.g., s_{pre}) to denote its value in a state.
(2) We distinguish between the initial and final values of an object by using an object identifier subscripted by *pre* to denote the object's initial value, and subscripted by *post* to denote its final value. Thus the assertion $s_{pre} = s_{post}$ says that the value of the object s is unchanged.
(3) The headers for a CLU procedure and a CLU procedure specification are intentionally similar. The only difference is that object identifiers, such as i, are introduced for returned objects in the header of a procedure specification. This is to provide a way to denote them in the assertions.
(4) The name of the used trait denotes the Shared Language component. The language of the assertions in the body of the procedure specification derives from the language of this Shared Language component.
(5) The **modifies at most** clause is an assertion that is given meaning as if it were conjoined to the postcondition (see above). It is syntactically separated from the precondition to highlight a procedure's potential side effect on the values of objects. It is an example of a *special assertion*; each interface language comes equipped with its own set of special assertions. They can be regarded as syntactic sugar for first-order assertions about state.

2.4 A Larch/CLU Cluster Specification

Figure 3 gives a Larch/CLU specification for a *set* cluster. It consists of a header, a link to its Shared Language component, and a body. The header consists of the type identifier, set, and a list of the procedure identifiers, pair, union, intersect, member, and size. Notice that set is the name of a type, not a sort. It is also the name of the cluster specification and is different from any trait name. The link from the interface component to the shared component is given by the used trait, SetOfE, and a **provides** clause. SetOfE supplies all sort and operator identifiers that appear in the assertions of the procedure specifications of the cluster specification. For example, the sort identifier, E, which appears in the postcondition of union, comes from SetOfE and is used for terms denoting integer values. The **provides** clause gives a mapping from the type identifier, set, to the sort identifier, SI, which also comes from SetOfE. This type-to-sort mapping determines the values over which set objects can range. All set objects are restricted to values denotable by terms of sort SI. The **provides** clause also indicates whether the type is mutable or not. The body of a cluster specification consists

```
set = cluster is pair, union, intersect, member, size
  uses SetOfE with [set for SI, int for E]
  provides mutable set from SI

    pair = proc (i, j: int) returns (s: set)
      ensures (s_post = add(add(empty, i_pre), j_pre)) ∧ new [s]
  end

  union = proc (s1, s2: set)
      modifies at most [s2]
      ensures ∀ j:E [has(s2_post, j) = (has(s1_pre, j) ∨ has(s2_pre, j))]
    end

  intersect = proc (s1, s2: set)
    modifies at most [s2]
    ensures ∀ j:E [has(s2_post, j) = (has(s1_pre, j) ∧ has(s2_pre, j))]
  end

  member = proc (s: set, i: int) returns (b: bool)
    ensures b_post = has(s_pre, i_pre)
  end

  size = proc (s: set) returns (i: int)
      ensures i_post = card(s_pre)
    end

  end set
```

Fig. 3. A set cluster specification.

of specifications of the procedures, which are of the form described for procedure specifications.

Three additional features of Larch/CLU are illustrated in the specification of pair: omitted **requires** clauses, omitted **modifies at most** clauses, and **new** assertions. First, the omission of a **requires** clause indicates that the precondition is true. Second, the omission of a **modifies at most** clause is the same as the explicit presence in the postcondition of the special assertion **modifies nothing**. This assertion states that no objects may be mutated by the procedure—for each call, the value of each object must be the same on return as on entry. Third, since CLU procedures can create new objects, we use **new** assertions to indicate what objects, if any, are created as a result of invoking a procedure satisfying the specification. For example, pair's specification states that it must not return a set object that existed when pair was invoked.

Let us consider writing a different set cluster specification, set2, that defines a different set type—one with a slightly different specification for the intersect procedure. Let the specification of set2 be the same as that of set in Figure 3, except that intersect2 returns the intersection of its two arguments only if they are not disjoint; otherwise it terminates exceptionally, signaling "disjoint." That is, let intersect2 be

```
intersect2 = proc (s1, s2: set) signals (disjoint)
  modifies at most [s2]
  ensures
    normally ∀j: E[has(s2_post, j) = (has(s1_pre, j) ∧ has(s2_pre, j))] except
    signals disjoint when ¬∃j: E [has(s1_pre, j) ∧ has(s2_pre, j)]
                  ensuring modifies nothing
end
```

Even though the two set cluster specifications, set and set2, specify different types, they both use the same trait, SetOfE. Therefore, set objects defined by set of Figure 3 range over values denoted by the same terms as set objects defined by set2. This difference illustrates that there is a clear distinction between a sort identifier and a type identifier. Although the trait SetOfE introduces the term empty of sort SI to denote the "empty" value, no object of type set2 will ever have such a value, since only nonempty set objects can be constructed by set2's (constructor) operations, pair, union, and intersect2.

An additional feature of Larch/CLU is illustrated by intersect2. CLU procedures may either terminate normally or terminate by signaling an exception. Larch/CLU uses the special assertions **returns** and **signals** *Signal Name* to denote both kinds of termination. Furthermore, Larch/CLU provides syntactic sugar for handling the many possible termination cases in a single assertion beginning with the keyword **normally**. Thus, the postcondition of intersect2 states that if s1 and s2 have no element in common, intersect2 raises the exception disjoint and modifies nothing. Otherwise, intersect2 returns normally and modifies s2 so that its final value is the intersection of the initial values of s1 and s2. Demarcating these individual cases enhances the readability of the specification and disciplines the specifier to consider all possible cases in a stylized way. More generally, an assertion of the form

normally *Normal Assn* **except**
signals *Signal Name*$_1$ **when** *Except Pre*$_1$ **ensuring** *Except Post*$_1$
...
signals *Signal Name*$_n$ **when** *Except Pre*$_n$ **ensuring** *Except Post*$_n$

is a shorthand for the assertion

(**returns** $\vee$ **signals** *Signal Name*$_1$ $\vee \cdots \vee$ **signals** *Signal Name*$_n$) $\wedge$
(**returns** $\Rightarrow$ (*Normal Assn* $\wedge \neg$(*Except Pre*$_1$ $\vee \cdots \vee$ *Except Pre*$_n$))) $\wedge$
(**signals** *Signal Name*$_1$ $\Rightarrow$ (*Except Pre*$_1$ $\wedge$ *Except Post*$_1$)) $\wedge$
$\cdots \wedge$
(**signals** *Signal Name*$_n$ $\Rightarrow$ (*Except Pre*$_n$ $\wedge$ *Except Post*$_n$))

2.5 Summary of the Language Features Illustrated

In summary, a Larch/CLU interface language component of a two-tiered specification includes

—a *header* that is similar or identical to that of a CLU program module;
—a *link* that names a *used trait* and provides type-to-sort mappings;
—a *body* that can contain
  first-order assertions based on the used trait;
  —special assertions: **new**, **modifies at most**, **modifies nothing**, **returns**, **signals**;
  —syntactic amenities like
    a separate **modifies at most** clause;
    default preconditions and modifies assertions indicated by omitted **requires** and **modifies at most** clauses;
    a stylized way to handle multiple termination conditions using the **normally** assertion.

3. INCREMENTALLY WRITING AN INTERFACE SPECIFICATION

As mentioned in the Introduction, writing Larch specifications is intended to occur during the design process with the help of machine support. In this section we illustrate how one writes an interface specification following Larch's two-tiered approach as intertwined with a typical top-down design process. We also mention some of the machine support a specifier might expect as a two-tiered specification is written.

3.1 Following the Approach

We sketch below a typical top-down design strategy that could be used in following the two-tiered approach.

—Develop an approximate intuition of the problem to be solved. This requires close, often verbal, interaction with the client who is posing the problem.
—Decide on the major abstractions.
  Interface language tier: Write the header information of the interface language components.
  Shared Language tier: Write the syntactic information of the Shared Language components of the specification (i.e., the sort identifiers and operator identifiers and signatures).
—Fill in the blanks.
  Interface language tier: Fill in the information in the bodies of the interface language components of the specification (e.g., write the assertions in the body of a procedure specification). Simultaneously generate additional operator and sort identifiers needed from the Shared Language components.
  Link between the two tiers: Define the explicit link to the Shared Language components of the specification.
  Shared Language tier: Fill in the semantic information in the bodies of the Shared Language components of the specification, namely, the theory of equality for terms.
—Check one's understanding of the problem and its formalization; repeat previous steps until they converge.

During this process of writing a specification, the specifier should also evaluate it for certain properties, such as consistency and completeness. Checking for these properties as a specification develops can increase one's confidence that a specification is on the right track. In the example of the next subsection, we describe a check for one such property, *totality*, to illustrate how feedback from evaluating a specification can influence the specifier. In Section 4 we discuss two other properties, *protection* and *nondeterminism*, which one might want to check of interface specifications.[3]

There are two points worth observing in regard to following the strategy outlined above, especially for large pieces of software. First, as with any overall design method, many iterations over the steps may be necessary. Writing a

[3] A more detailed discussion of these and other properties of interface specifications can be found in [51] and [52].

specification sharpens a specifier's intuition of the problem. Hidden design decisions surface. Addressing postponed decisions often requires modifications of decisions made earlier. Second, the specifier should be willing to discard large chunks of a specification in the process of refining the abstractions. This is especially true after the first iteration. Often, after a large investment in time and effort, the specifier (or designer or programmer) is reluctant to start anew or to try alternate tactics. With sufficient machine support the specifier should be able to save time and effort often spent in managing and maintaining the consistency of a large specification.

3.2 Following the Two-Tiered Approach on an Example

In this section we trace through one iteration of the strategy outlined in the previous section by presenting a series of snapshots to show the incremental development of a specification. We choose a simple example to avoid having too many details get in the way of the points we wish to make.

Suppose we want to write a specification of a dictionary that contains the definitions of words and that can be used to check the spelling of words. For simplicity, let us assume that a word can appear only once in a dictionary, and each word has exactly one definition. Furthermore, if a word is not in the dictionary, then the word is either misspelled or unknown to the dictionary (e.g., a rarely used word might not be found in an abridged dictionary). Intuitively, a dictionary is like a table that stores key-value pairs, where words are the keys and definitions are the values.

From this informal description of a dictionary and an intuitive understanding of its usage, we next have to decide on the major abstractions. We choose the data types of interest to be dictionary, word, and definition. Therefore, we need to write cluster specifications for each of the three types and appropriate traits for the values of objects of each type. Since we need a used trait for each cluster specification, let us name them DictVals, WordVals, and DefVals. Figure 4 depicts the situation so far. We are presuming the use of a syntax-directed specification editor that displays the templates shown in the figure and prompts us to fill in each "...".

We begin by further developing the dictionary cluster specification and the corresponding DictVals trait, and postpone developing the other specification components until later. Given the informal description of the usage of a dictionary, we have to decide what operations would most likely be performed on dictionaries. Some of the table-like operations we might want to perform are to create a dictionary, add a new word and its definition to a dictionary, get the definition of a word, and check to see if a word is in a dictionary. After filling in some syntactic information for dictionary, we have the situation as shown in Figure 5. Visible changes from one snapshot to the next are shown in italics.

Next we start filling in the bodies of the procedure specifications, and simultaneously generate sort and operator identifiers that must be supplied by DictVals. We start with create. We do not want any restrictions on the computation state in creating a new dictionary, nor do we want any objects to be mutated; we want the value of the returned dictionary to be empty, and we want the dictionary itself to be some new object. So for create we have (notice the

```
dictionary = cluster is ...                DictVals: trait
  uses DictVals ...                          introduces
  provides dictionary from ...                 ...
    ...                                      constrains
  end dictionary                               ...
word = cluster is ...                      WordVals: trait
  uses WordVals ...                          introduces
  provides word from ...                       ...
    ...                                      constrains
  end word                                     ...
definition = cluster is ...                DefVals: trait
  uses DefVals ...                           introduces
  provides definition from ...                 ...
    ...                                      constrains
  end definition                               ...
```

(a) (b)

Fig. 4. Dictionary specification: Snapshot 1. (a) Interface language components; (b) Shared Language components.

```
dictionary = cluster is create, add_word, get_definition, check_word
  uses DictVals ...
  provides dictionary from ...
    create = proc () returns (d: dictionary)
      requires ...
      modifies at most ...
      ensures ...
    end
    add_word = proc (d: dictionary, w: word, def: definition)
      requires ...
      modifies at most ...
      ensures ...
    end
    get_definition = proc (d: dictionary, w: word) returns (def: definition)
      requires ...
      modifies at most ...
      ensures ...
    end
    check_word = proc (d: dictionary, w: word) returns (b: bool)
      requires ...
      modifies at most ...
      ensures ...
    end
  end dictionary
DictVals: trait
  introduces
    ...
  constrains
    ...
```

Fig. 5. Dictionary specification: Snapshot 2.

```
dictionary = cluster is create, add_word, get_definition, check_word
  uses DictVals with [dictionary for D]
  provides dictionary from D
    create = proc () returns (d: dictionary)
      ensures (d_post = empty) ∧ new [d]
    end
    add_word = proc (d: dictionary, w: word, def: definition)
      requires . . .
      modifies at most . . .
      ensures . . .
    end
    get_definition = proc (d: dictionary, w: word) returns (def: definition)
      requires . . .
      modifies at most . . .
      ensures . . .
    end
    check_word = proc (d: dictionary, w: word) returns (b: bool)
      requires . . .
      modifies at most . . .
      ensures . . .
    end
  end dictionary
DictVals: trait
  introduces
    empty: → D
      . . .
  constrains
    . . .
```

Fig. 6. Dictionary specification: Snapshot 3.

deletion of the **requires** and **modifies at most** clauses):

```
create = proc () returns (d: dictionary)
  ensures (d_post = empty) ∧ new [d]
end
```

In order to denote the empty value of a dictionary, we use the operator identifier, empty, in create's postcondition. The empty operator must be defined by DictVals by first giving empty a signature, which in turn causes us to introduce a sort identifier (e.g., D) to which the type identifier dictionary can map. Consequently, we can define the type-to-sort mapping in the **uses** and **provides** clauses of dictionary. We now have the situation shown in Figure 6.

Next we turn to filling in the body of add_word. We want to add a word and its definition to a dictionary only if the word is not already in the dictionary. We state this constraint in the precondition of add_word. We have

```
add_word = proc (d: dictionary, w: word, def: definition)
  requires ¬isIn(d_pre, w_pre)
  modifies at most [d]
  ensures d_post = insert(d_pre, w_pre, def_pre)
end
```

```
dictionary = cluster is create, add_word, get_definition, check_word
  uses DictVals with [dictionary for D, word for W, definition for Dfn]
  provides mutable dictionary from D
    create = proc () returns (d: dictionary)
      ensures (d_post = empty) ∧ new [d]
    end
    add_word = proc (d: dictionary, w: word, def: definition)
      requires ¬ isIn(d_pre, w_pre)
      modifies at most [d]
      ensures d_post = insert(d_pre, w_pre, def_pre)
    end
    get_definition = proc (d: dictionary, w: word) returns (def: definition)
      requires . . .
      modifies at most . . .
      ensures . . .
    end
    check_word = proc (d: dictionary, w: word) returns (b: bool)
      requires . . .
      modifies at most . . .
      ensures . . .
    end
  end dictionary
DictVals: trait
  introduces
    empty: → D
    insert: D, W, Dfn → D
    isIn: D, W → Bool
      . . .
  constrains empty, insert, isIn so that
    for all [d: D, w, w1: W, dfn: Dfn]
      isIn(empty, w) = false
      isIn(insert(d, w, dfn), w1) = (w = w1) ∨ (isIn(d, w1))
        . . .
```

Fig. 7. Dictionary specification: Snapshot 4.

Notice a design decision we have made: by allowing the dictionary input to add_word to be possibly mutated, we have decided to make dictionary a mutable type. We document this decision in the **provides** clause of the dictionary with the keyword **mutable**.

The definitions of the operators isIn and insert are still pending in DictVals. To give a signature for insert, we introduce sort identifiers W and Dfn, corresponding to the types word and definition, respectively. Thus we can refine the specifications of the types word and definition in Figure 4 by completing their **provides** clauses. We can also write equations in DictVals to define the operators already introduced. Figure 7 shows the situation so far. We show only the dictionary cluster specification, so we cannot show the changes to the **provides** clauses of the word and definition cluster specifications.

Continuing this process by filling in the bodies of get_definition and check_word causes us to introduce only one more operator identifier, lookup.

```
dictionary = cluster is create, add_word, get_definition, check_word
  uses DictVals with [dictionary for D, word for W, definition for Dfn]
  provides mutable dictionary from D
    create = proc () returns (d: dictionary)
      ensures (d_post = empty) ∧ new [d]
    end
    add_word = proc (d: dictionary, w: word, def: definition)
      requires ¬ isIn(d_pre, w_pre)
      modifies at most [d]
      ensures d_post = insert(d_pre, w_pre, def_pre)
    end
    get_definition = proc (d: dictionary, w: word) returns (def: definition)
      requires isIn(d_pre, w_pre)
      ensures def_post = lookup(d_pre, w_pre)
    end
    check_word = proc (d: dictionary, w: word) returns (b: bool)
      ensures b_post = isIn(d_pre, w_pre)
    end
  end dictionary
DictVals: trait
  introduces
    empty: → D
    insert: D, W, Dfn → D
    isIn: D, W → Bool
    lookup: D, W → Dfn
  constrains empty, insert, isIn, lookup so that
    for all [d: D, w, w1: W, dfn: Dfn]
      isIn(empty, w) = false
      isIn(insert(d, w, dfn), w1) = (w = w1) ∨ (isIn(d, w1))
      lookup(insert(d, w, dfn), w1) = if w = w1 then dfn else lookup(d, w1)
```

Fig. 8. Dictionary specification: Snapshot 5.

After adding an equation to define lookup in DictVals, we end up with a dictionary specification and a DictVals trait as shown in Figure 8.

3.2.1 *Evaluating the Dictionary Cluster Specification So Far.* At this point, before proceeding to the word and definition cluster specifications, it is worth reflecting on the dictionary specification we have just written. As mentioned in our design strategy, during the incremental development of a specification, it is useful to evaluate a specification to see if it can be improved and to assure us that we are on the right track. In this section we discuss the evaluation of interface specifications for the property *totality*.

Notice that the precondition of the add_word specification is not (identically) true, which means that the behavior of an add_word procedure is left unspecified for some possible states in which it can be invoked. We say the add_word specification is not *total* [51]. Upon checking add_word for totality, we may be inclined to make it total and handle the case for which the word we attempt to

add to the dictionary is already in the dictionary. We might modify add_word to terminate exceptionally in this case:

```
add_word = proc (d: dictionary, w: word, def: definition)
                    signals (alreadyIn)
  modifies at most [d]
  ensures
    normally d_post = insert(d_pre, w_pre, def_pre) except
    signals alreadyIn when isIn(d_pre, w_pre) ensuring modifies nothing
end
```

Similarly, if we were to check get_definition for totality, we would find that it also is not total. We choose to make it total, and handle the case in which we attempt to get the definition of a word that is not in the dictionary:

```
get_definition = proc (d: dictionary, w: word) returns (def: definition)
                          signals (wordNotIn)
  ensures
    normally def_post = lookup(d_pre, w_pre) except
    signals wordNotIn when ¬isIn(d_pre, w_pre)
end
```

If we were to decide to leave a procedure specification not total, then the implementer would be free to choose what the behavior of the procedure would be in any unspecified cases. Unfortunately, implementers may often forget to handle unspecified cases, which may lead to surprising or erroneous behavior. On the other hand, it may not be necessary to handle unspecified cases if one is sure that they will never arise. For example, the choose procedure specification of Figure 2 is not total. If it were defined to operate on sets as defined by the set2 cluster specification described in Section 2.4, there would be no need to handle the "isEmpty" case, since it would never arise (assuming a correct implementation of set2).

3.2.2 *Completing the Remaining Interface Specifications.* We now turn to filling in the blanks for the word and definition cluster specifications and the WordVals and DefVals traits. Recall that the informal description of the usage of a dictionary requires that we must be able to check the spelling of a given word against the spellings of the words in the dictionary. This requirement implies that the word cluster must have a procedure that tests for equality between two words. No other requirements or constraints were made on words, such as if words are sequences of only alphabetic characters (perhaps numerals and punctuation symbols are allowed) or if there exists a "null" word. Therefore, until further constraints are made by the client, it suffices to include in the word cluster specification simply a specification of an equal procedure.

Finally, we turn to definition and DefVals. We have even less information about definitions of words in a dictionary than we have about words. For instance, we do not know whether definitions are sentences, phrases, or combinations of both, or whether they must conform to a fixed format. The only information we can include in the definition cluster specification is the type-to-sort mapping in the **provides** clause. Recall that we generated this information when we introduced the insert operator for the dictionary cluster specification.

We have essentially gone through one iteration of the strategy as outlined in Section 3.1. At this point, we need to return to the client and ask for more information. After further elaboration of the problem description, appropriate additions and modifications can then be made to the specification.

4. IMPLICATIONS OF THE TWO-TIERED APPROACH

In this section we explore the consequences of the two-tiered approach. First, in Section 4.1, we discuss the interaction between the Shared Language and interface language components of a Larch specification; then, in Section 4.2, we discuss how Larch interface languages support the two-tiered approach in general.

4.1 Interaction between the Two Tiers

The Larch style of specification presumes a style of program design in which abstract data types play a prominent role. One of the difficulties people may face in writing a two-tiered specification of an abstract data type (e.g., a Larch/CLU cluster specification) is figuring out where to put what. If the interesting abstractions are defined in the Shared Language components, interface language components of specifications of abstract data types can appear to be trivial. This triviality of interface language components is in the spirit of the Larch style, since we would rather place the complexity of a two-tiered specification in the Shared Language component (the programming language-independent part) and keep the interface language component simple. So, for example, a more typical cluster specification for sets, which uses the SetOfE trait, than either presented in Section 2.4 would define operations on sets to be create, insert, delete, member, and size. In this case the postconditions for create, insert, and delete would refer to the trait operators empty, add, and remove, respectively, in an obvious way. As a guideline to specifiers, therefore, in order to keep interface language components simple, if the pre- and postconditions appear complicated, then very likely the used trait is not providing the appropriate abstractions and needs redefinition.

Another distinction between interface and Shared Language components is that interface specifications describe what is to be implemented; traits do not. In particular, operations of abstract data types defined through procedure specifications are intended to be implemented by procedures (of a program), but operators of traits are not. Trait operators are considered as auxiliary, or hidden, functions of the specification. Thus, for example, the pair operation of the set type as specified in Figure 3 is to be implemented by some CLU procedure, but the add operator of the SetOfE trait is not. More specifically, if the CLU implementation of the set abstraction uses arrays to represent sets, then the implementation of the pair operation would very likely make two calls to the array[int]$addh procedure; it would be nonsensical for the implementation to make two calls to a set$add procedure since none would exist.[4]

Two properties of interface specifications illustrate two other ways the two tiers interact. The first property, *protection*, is related to the sufficient

[4] In CLU syntax, *type_name*$*op_name* denotes the operation *op_name* of the type *type_name*. For CLU arrays, array[int]$addh($a$, i) adds the integer i to the high end of the array a.

completeness of an algebraic specification [17]. The Larch Shared Language does not require traits to be sufficiently complete and provides a construct **exempts** for indicating that the meaning of certain terms has been intentionally unconstrained; hence, an **exempt** term indicates an intentional incompleteness in a trait. By partitioning a specification into two tiers, we can avoid in interface language components such incompletenesses in Shared Language components. When writing an interface specification whose used trait is not sufficiently complete, nontrivial preconditions can be used to ensure that the interface is *protected* from the incompletenesses of its used trait (i.e., that its meaning does not depend on the meaning of any **exempt** terms). For example, the DictVals trait is not sufficiently complete because the meaning of lookup(empty, w) is left unconstrained. If the get_definition procedure specification of the dictionary cluster specification were as follows,

get_definition = **proc** (d: dictionary, w: word) **returns** (def: definition)
  requires isIn(d_{pre}, w_{pre})
  ensures def_{post} = lookup(d_{pre}, w_{pre})
end

then, because of its precondition, the meaning of get_definition is independent of the meaning of lookup(empty, w). In this case we say dictionary is *protective* of its used trait. If the precondition were "true" and the postcondition remained as above, then dictionary would not be protective of DictVals.

Factoring a specification into two tiers allows us to factor our checks as well. If upon checking a trait for sufficient completeness we discover it is not, we may be inclined to check the interface language component for protection. Notice a typical chain of events that may occur through the feedback gained from checking: a check for incompleteness in a trait may lead to a check for protection of the interface component, which may lead to introducing a nontrivial precondition; then a check for totality (Section 3.2.1) may lead to introducing the handling of normal and exceptional cases, which may lead to the use of the special **normally** assertion.

A second property, *nondeterminism*, of interface specifications deals with a different kind of incompleteness—that of underconstraining, typically intentionally, final values of objects as specified in postconditions. Nondeterminism can manifest itself in interface specifications in at least two ways: the presence of a **modifies at most** clause allows for a possible, but not required, change of the values of input arguments; and the postcondition may allow for the final value of an object to range over a set of final values. For example, the specification of choose of Figure 2 is nondeterministic because the value of the integer returned may be different in one invocation of choose from another. If the second conjunct in choose's postcondition were removed, then choose would be nondeterministic for an additional reason: whether or not the final value of s is changed (the **modifies at most** clause allows for s's value to change) is a choice left up to the implementer.

Nondeterminism cannot be introduced by traits. The mathematical basis of algebra and of the Larch Shared Language depends on the ability to freely substitute "equals for equals." This property would be destroyed if trait operators

were allowed to be nondeterministic functions. Furthermore, nondeterminism in an interface should not be confused with incompleteness in a trait. Although trait operators may be introduced without giving enough axioms to define them fully (e.g., as in DictVals), it is always the case that for every term t, $t = t$ (e.g., lookup(empty, w) = lookup(empty, w)).

4.2 Language Support for Approach

The Larch family of specification languages supports the two-tiered approach in many ways. We summarize four below.

Of primary importance, the separation of concerns is evident in the assertion language used in the bodies of procedure specifications. Two points to observe about the assertion language are that (1) Shared Language components contain only state-independent assertions; interface language components contain state-dependent assertions. All special assertions (e.g., **new**, **modifies**, and **signals**) are state-dependent, and are found in interface language components only. These special assertions obtain their meaning from the semantics of the target programming language and are introduced to highlight certain important classes of state transformations of the programming language. (2) By explicitly including a Shared Language component in an interface specification, we gain the advantage that every symbol in an assertion is precisely defined within a specification. In some other specification methods [27, 44], there is a reliance on an interpretation for symbols in an assertion, where the interpretation comes from outside the specification. For example, the meanings of symbols like $\in$ and $\subseteq$ might come from textbooks on set theory. In contrast, some other methods (e.g., Jones's and SPECIAL's, as mentioned in Section 1.4) provide an assertion language defined within the specification, but restrict the symbols to come from a fixed set of primitives. We gain the advantage that the user is able to provide just the symbols necessary to write the assertions in the body of an interface specification.

Second, the separation allows the interface language component to serve as a buffer between the programming language and the Shared Language. For example, the CLU concepts of an object and its type are kept separate from the Shared Language concepts of a term, which denotes a value, and its sort. Since objects of two different types can range over the same set of values, and hence have values denotable by the same sort, more than one type can map to the same sort. The **provides** clause that appears in a cluster specification specifies to what sort a type maps.

Third, although the Larch Shared Language is a proper subset of a Larch interface language, the Shared Language component of a two-tiered specification is syntactically separate from the interface language component. They are connected by naming the Shared Language component via the **uses** clause (and **provides** clause for cluster specifications). This separation allows us to keep individual components of a specification relatively small. We encapsulate a few key concepts into a single small specification module rather than many concepts into a single large specification. Understanding each small component should be easier than understanding one large specification. Furthermore, reuse of Shared Language components is encouraged. For example, the DictVals trait of the dictionary cluster specification could be used for defining a symbol table cluster

specification (or vice versa, if a TableVals trait had existed, it could have been used for the dictionary cluster specification).

Finally, since different interface languages all derive part of their meaning from the Larch Shared Language, defining a new interface language requires only that that subset of the language built on top of the Larch Shared Language be defined. Additionally, we can use the same Shared Language components for interface language components written in different interface languages.

5. FURTHER WORK

We expect this work to be explored further in the following three directions: developing other Larch interface languages, building machine support, and applying the two-tiered approach to examples.

One test of the two-tiered approach is to develop Larch interface languages for other programming languages, both sequential and concurrent. Development of interface languages for other sequential programming languages has been done informally for Cedar Mesa [29] and Pascal [22], and is being explored for Modula-2. From this experience, we offer the following suggestions for designing a new interface language.

—Start with the basic skeleton of Larch/CLU.
—Define the target programming language's model of state. Much of this can be done through traits.
—Identify which components of state may change. Introduce special assertions to highlight those state changes that are anticipated to occur frequently.
—Consider other semantic issues of the programming language, and decide how and where to handle them, that is, at which level (traits or interfaces). Issues to consider are the storage model (heap versus stack), termination and error handling (are errors handled and propagated, or do they abort the program?), and parameter-passing mechanisms (CLU uses call-by-sharing, whereas Pascal has both call-by-value and call-by-reference).

One consequence of separating a specification into two tiers is that there is more to manage: more pieces and more possible interaction between pieces. The development of sophisticated machine aids addresses the need to help the human keep track of a growing specification. Without machine support, we have no hope of expecting either specifiers to write or programmers to use specifications, except as an academic exercise. Minimally, machine support should provide ways to manage the text of specifications; ideally, it should provide ways to reason about their meaning as well. The Larch Project has been designing and implementing software tools as part of a specification environment. Included in this development effort are implementations of a syntax and static-semantics checker for the Larch Shared Language [35] and a semantic checker that can manipulate equations [13, 36], and the design of a syntax-directed editor [54] and a specification library [3, 25]. Throughout this paper, we have presumed the existence of some of these tools. For example, for the set and set2 specifications, we presumed the existence of a trait library from which we borrowed the SetOfE trait; in the dictionary specification, we presumed that no appropriate trait was

available, and imagined the use of a highly interactive editor to help us write the DictVals trait.

Finally, the two-tiered approach needs to be tested on realistic examples of substantial size. By trying it out, we can evaluate whether the two-level partitioning is good, whether it makes it easier to read and write specifications, and whether it leads to better specifications. We can also see whether the separation of concerns leads to a better understanding of what it is we are trying to specify. With more experimentation, we hope to show the utility of using formal specifications; in particular, to demonstrate that forcing precision in the design process has a beneficial effect on the overall programming process.

ACKNOWLEDGMENTS

I would like to thank John Guttag and Jim Horning for the opportunity to work with them on the Larch Project. They have greatly influenced my ideas about formal specifications in general, and about Larch specification languages in particular. I would also like to thank John Guttag, Sharon Anderson, and the referees for their helpful comments on drafts of earlier versions of this paper.

REFERENCES

1. ABRIAL, J. R. The specification language Z: Syntax and semantics. Programming Research Group, Oxford University, 1980.
2. GOGUEN, J. A., THATCHER, J. W., WAGNER, E. G., AND WRIGHT, J. B. Abstract data types as initial algebras and correctness of data representations. In *Proceedings of the Conference on Computer Graphics, Pattern Recognition and Data Structures* (May 1975), ACM, New York, 89–93.
3. ATREYA, S. K. Formal specification of a specification library. M.S. thesis, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Mass., May 1982.
4. BERZINS, V. Abstract model specifications for data abstractions. MIT Laboratory for Computer Science, Cambridge, Mass., July 1979.
5. BIRKHOFF, G., AND LIPSON, J. D. Heterogeneous algebras. *J. Comb. Theor. 8* (1970), 115–133.
6. BJORNER, D., AND JONES, C. G. (Eds.). *The Vienna Development Method: The Meta-language. Lecture Notes in Computer Science, 61.* Springer, New York, 1978.
7. BURSTALL, R. M., AND GOGUEN, J. A. Putting theories together to make specifications. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence* (Aug. 1977), 1045–1058. Invited paper.
8. BURSTALL, R. M., AND GOGUEN, J. A. An informal introduction to specifications using CLEAR. In *The Correctness Problem in Computer Science*, 1981.
9. CAINE, S. H., AND GORDON, E. K. PDL—A tool for software design. In *Proceedings of the 1975 National Computer Conference* (Anaheim, Calif., May 19–22). AFIPS Press, Reston, Va., 1975, 271–276.
10. EHRICH, H.-D. Extensions and implementations of abstract data type specifications. In *Mathematical Foundations of Computer Science 1978, Proceedings. Lecture Notes in Computer Science, 64.* Springer, New York, 1978, 155–164.
11. EHRIG, H., KREOWSKI, H.-J., THATCHER, J., WAGNER, E., AND WRIGHT, J. Parameterized data types in algebraic specification languages. In *Automata, Languages, and Programming. Lectures Notes in Computer Science, 85.* Springer, New York, 1980, 157–168.
12. EHRIG, H., AND MAHR, B. *Fundamentals of Algebraic Specification 1.* Springer, New York, 1985.
13. FORGAARD, R. A program for generating and analyzing term rewriting systems. M.S. thesis, TR-343, MIT Laboratory for Computer Science, Cambridge, Mass., 1985.
14. GOGUEN, J. A. Abstract errors for abstract data types. In *Proceedings of the IFIP Working Conference on Formal Basis of Programming Concepts* (Aug. 1977). IFIP, 21.1–21.32.

15. GOGUEN, J. A., AND PARSAYE-GHOMI, K. Algebraic denotational semantics using parameterized abstract modules. CSL-119, Stanford Research Institute, Menlo Park, Calif., Feb. 1981.
16. GOOD, D. I., COHEN, R. M., HOCH, C. G., HUNTER, L. W., AND HARE, D. F. Report on the language Gypsy, version 2.0. ICSCA-CMP-10, Certifiable Minicomputer Project, Univ. of Texas, Austin, Sept. 1978.
17. GUTTAG, J. V. The specification and application to programming of abstract data types. Ph.D. dissertation, Univ. of Toronto, Toronto, Sept. 1975.
18. GUTTAG, J. V. Abstract data types and the development of data structures. *Commun. ACM 20*, 6 (June 1977), 396–404.
19. GUTTAG, J. V., AND HORNING, J. J. Formal specification as a design tool. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages* (Las Vegas, Nev., Jan. 1980), ACM, New York, 251–261.
20. GUTTAG, J. V., HORNING, J. J., AND WING, J. M. Some notes on putting formal specifications to productive use. *Sci. Comput. Program. 2*, 1 (Oct. 1982), 53–68.
21. GUTTAG, J. V., AND HORNING, J. J. An introduction to the Larch Shared Language. In *Proceedings of the IFIP 9th World Computer Congress* (Paris, Sept. 1983), IFIP, 1983.
22. GUTTAG, J. V., HORNING, J. J., AND WING, J. M. The Larch family of specification languages. *IEEE Softw. 2*, 5 (Sept. 1985), 24–36.
23. GUTTAG, J. V., HORNING, J. J., AND WING, J. M. Larch in five easy pieces. 5, DEC Systems Research Center, July 1985.
24. GUTTAG, J. V., AND HORNING, J. J. Report on the Larch Shared Language. *Sci. Comput. Program. 6* (1986), 103–134.
25. GUTTAG, J. V., AND HORNING, J. J. A Larch Shared Language handbook. *Sci. Comput. Program. 6* (1986), 135–157.
26. HOARE, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969), 576–583.
27. HOARE, C. A. R. Proof of correctness of data representations. *Acta Inf. 1*, 1 (1972), 271–281.
28. HORNING, J. J. Combining algebraic and predicative specifications in Larch. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development, TAPSOFT, 1985. Lecture Notes in Computer Science, 186.* Springer, New York, 1985, 12–26.
29. HORNING, J. J. Cedar Mesa interface language. Private communication, 1983.
30. JACKSON, M. A. *Principles of Program Design.* Academic Press, London, 1975.
31. JONES, C. B. *Software Development: A Rigorous Approach.* Prentice-Hall International, Englewood Cliffs, N.J., 1980.
32. KAMIN, S. Final data types and their specification. *ACM Trans. Program. Lang. Syst. 5*, 1 (Jan. 1983), 97–121.
33. KAPUR, D. Towards a theory for abstract data types. TR-237, MIT Laboratory for Computer Science, Cambridge, Mass., May 1980.
34. KATZAN, H., JR. *Systems Design and Documentation: An Introduction to the HIPO Method.* Van Nostrand Reinhold, New York, 1976.
35. KOWNACKI, R. W. Semantic checking of formal specifications. M.S. thesis, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Mass., Aug. 1984.
36. LESCANNE, P. Computer experiments with the REVE term rewriting system generator. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages* (Austin, Tex., Jan. 1983). ACM, New York, 1983, 99–108.
37. LISKOV, B. H., AND ZILLES, S. N. Specification techniques for data abstractions. *IEEE Trans. Softw. Eng. SE-1*, 1 (1975), 7–19.
38. LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (Aug. 1977), 564–576.
39. LISKOV, B. H., AND BERZINS, V. An appraisal of program specifications. Research Directions in Software Technology, Cambridge, Mass., 1979.
40. LISKOV, B. H., ET AL. *CLU Reference Manual: Lecture Notes in Computer Science, 114.* Springer, New York, 1981.
41. MUSSER, D. R. Abstract data type specification in the Affirm system. *IEEE Trans. Softw. Eng. 6*, 1 (Jan. 1980), 24–32.
42. NAKAJIMA, R., HONDA, M., AND NAKAHARA, H. Hierarchical program specification and verification—A many-sorted logical approach. *Acta Inf. 14* (1980), 135–155.

43. NAKAJIMA, R., AND YUASA, T. *The Iota Programming System.* Springer, New York, 1983.
44. PARNAS, D. L. A technique for software module specification with examples. *Commun. ACM 15*, 5 (May 1972), 330–336.
45. ROBINSON, L., AND ROUBINE, O. SPECIAL—A specification and assertion language. CSL-46, Stanford Research Institute, Menlo Park, Calif., Jan. 1977.
46. SCHEID, J., AND ANDERSON, S. The Ina Jo specification language reference manual. TM-(L)-6021/001/00, System Development Corp., Santa Monica, Calif., Mar. 1985.
47. STANDISH, T. A. Data structures: An axiomatic approach. Rep. 2639, Bolt, Beranek, and Newman, Cambridge, Mass., Aug. 1973.
48. SUFRIN, B., MORGAN, C., SORENSEN, I., AND HAYES, I. Notes for a Z handbook: Part 1—The mathematical language. Programming Research Group, Oxford Univ., Computing Laboratory, Aug. 1984.
49. THATCHER, J. W., WAGNER, E. G., AND WRIGHT, J. B. Data type specification: Parameterization and the power of specification techniques. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing* (May), ACM, New York, 1978, 119–132.
50. WAND, M. Final algebra semantics and data type extensions. *J. Comput. Syst. Sci. 19*, 1 (Aug. 1979), 27–44.
51. WING, J. M. A two-tiered approach to specifying programs. MIT-LCS-TR-299, MIT Laboratory for Computer Science, Cambridge, Mass., June 1983.
52. WING, J. M. Helping specifiers evaluate their specifications. In *Proceedings of the 2nd Software Engineering Conference* (June 1984). AFCET, 179–191.
53. YOURDON, E., AND CONSTANTINE, L. L. *Structured Design: Fundamentals of a Discipline of Computer Programs and Systems Design*, 2nd ed. Yourdon Press, New York, 1978.
54. ZACHARY, J. A syntax-directed tool for constructing specifications. M.S. thesis, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Mass., Mar. 1983.
55. ZAVE, P. An operational approach to requirements specification for embedded systems. *IEEE Trans. Softw. Eng. 8*, 3 (May 1982), 250–269.
56. ZILLES, S. N. Abstract specifications for data types. IBM Research Laboratory, San Jose, Calif., 1975.

Received December 1983; revised April 1985 and April 1986; accepted April 1986