IEEE Software

The Larch Family of Specification Languages

John V. Guttag, Massachusetts Institute of Technology
James J. Horning, Digital Equipment Corporation
Jeannette M. Wing, Carnegie-Mellon University

The Larch Family of Specification Languages

John V. Guttag, Massachusetts Institute of Technology
James J. Horning, Digital Equipment Corporation
Jeannette M. Wing, Carnegie-Mellon University

Larch specifications are two-tiered. Each one has a component written in an algebraic language and another tailored to a programming language.

The use of suitable formalisms in the specification of computer programs and parts of computer programs offers significant advantages.[1] Although there is considerable theoretical interest in this area,[2] practical experience is rather limited. The Larch Project, a research project intended to have practical applications in the next few years, is developing tools and techniques to aid in the productive application of formal specifications. A major part of the project is a family of specification languages. Each specification has components written in two languages. The Larch interface languages are particular to specific programming languages, while the Larch Shared Language is common to all languages.

Some important aspects of the Larch family of specification languages are

- *Composability.* The Larch languages are designed for the incremental construction of specifications from other specifications.
- *Emphasis on presentation.* The Larch languages are designed to be readable. Among other things, Larch's composition mechanisms are defined as operations on specifications, rather than on theories or models.[3]
- *Suitability for integrated interactive tools.* The Larch languages are designed to facilitate the interactive construction and incremental checking of specifications.
- *Semantic checking.* The Larch languages are designed to enable extensive checking of specifications as they are being constructed. An important aspect of our approach is the use of a powerful theorem prover for semantic checking to supplement the syntactic checking commonly defined for specification languages.
- *Localized programming language dependencies.* Each Larch interface language encapsulates the features needed to write concise and comprehensible specifications for a particular programming language and incorporates Larch Shared Language specifications in a uniform way.

The Larch interface languages specify program modules, providing information needed to write programs that use these modules (Figure 1).[4] A critical part of the interface is how the module communicates with its environment, yet communication mechanisms differ from programming language to programming language, sometimes in subtle ways. We have found it easier to be precise about communication when the specification language reflects the programming language. Such specifications are generally shorter than those written in a "universal" interface language. They also seem clearer to programmers who implement modules and to programmers who use them.

Each Larch interface language deals with what can be observed about the behavior of programs written in a particular programming language. It provides a way to write assertions

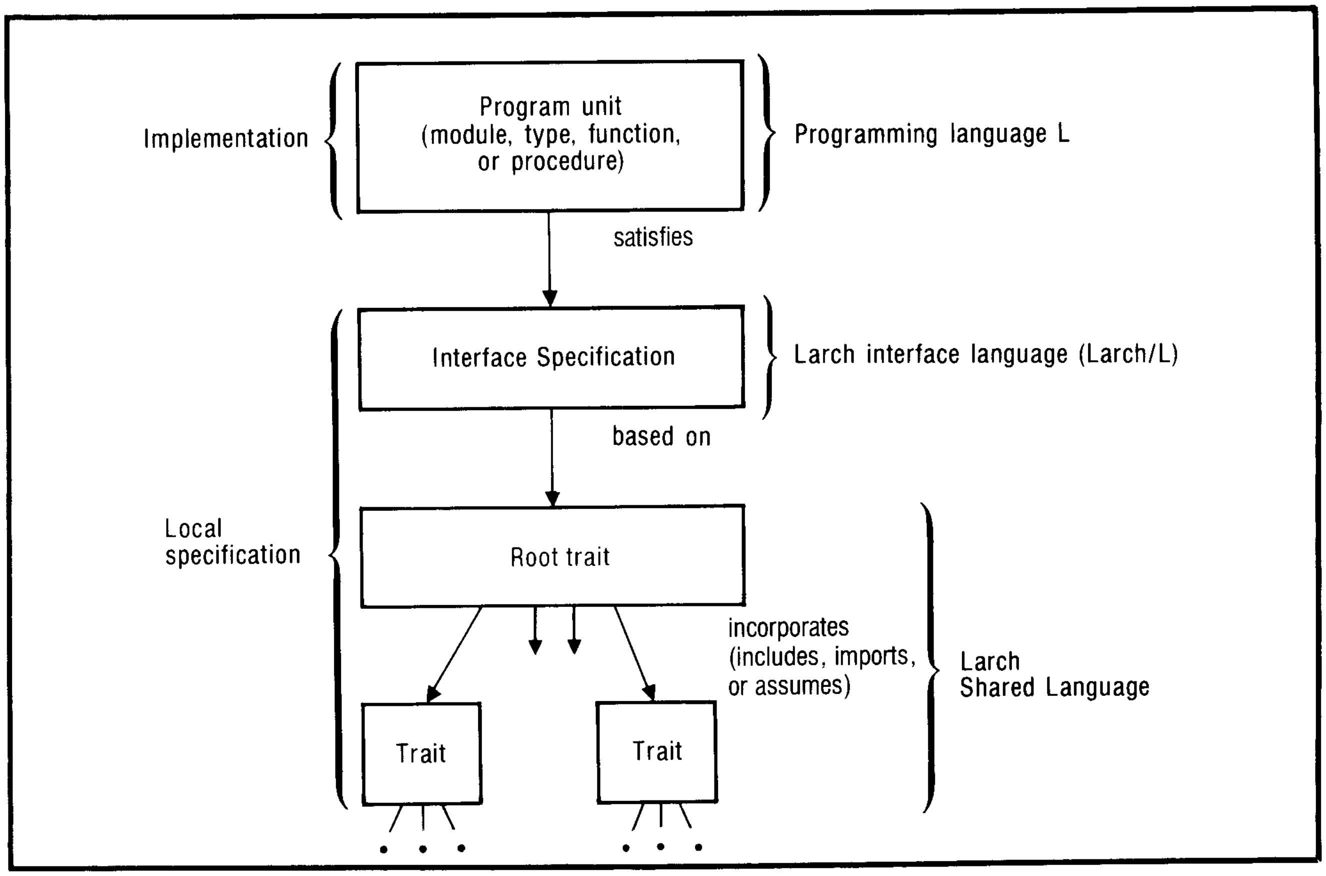


Figure 1. Two-tiered specification in Larch.

about program states. It incorporates programming-language-specific notions for constructs such as side effects, exception handling, and iterators, and its simplicity or complexity depends largely upon the simplicity or complexity of the observable state and state transformations of its programming language.

The Larch Shared Language is used to define terms used in interface specifications, by generating theories that are independent of any programming language. The Larch Shared Language is primarily algebraic: equations define relations among operators, giving meaning to the notion of equality among terms that appear in interface specifications.

Larch is intended to support a style of program design in which data abstractions play a prominent role.[5] Each Larch interface language has a mechanism for specifying data abstractions. If its programming language provides direct support for data abstractions, the Larch interface language facility is modeled on that of the programming language; if it does not, the facility is designed to be compatible with other aspects of the programming language.

The Larch Shared Language

This is not the place for a complete description of either the syntax or the semantics of the Larch Shared Language. Both have been published elsewhere (see bibliography in box on p. 31). Instead, we present a series of short examples that introduce virtually the entire language a few features at a time.

The *trait* is the basic unit of specification in the Larch Shared Language. A trait introduces operators and specifies their properties. Sometimes the collection of operators will correspond to an abstract data type. Frequently, however, it is useful to define properties that do not fully characterize a type.

Our first example (below) is a trait specifying a class of tables that store values in indexed places. It is similar to a conventional algebraic specification in the style of Guttag and Horning[6] or Ehrig and Mahr.[7]

```
TableSpec: trait
  introduces
    new: → Table
    add: Table, Index, Val → Table
    # ∊ # : Index, Table → Bool
    eval: Table, Index → Val
    isEmpty: Table → Bool
    size: Table → Card
  constrains new, add, ∊, eval, isEmpty,
      size so that
    for all [ind, indl: Index,
      val: Val, t: Table]
      eval(add(t, ind, val), indl) =
        if ind = indl
        then val
        else eval(t, indl)
      ind ∊ new = false
      ind ∊ add(t, indl, val) =
        (ind = indl) | (ind ∊ t)
      size(new) = 0
      size(add(t, ind, val)) = if ind ∊ t
        then size(t) else size(t) + 1
      isEmpty(t) = (size(t) = 0)
```

Kinds of specifications

We find it helpful to classify specifications and specification languages in a variety of ways, each of which has important consequences for the development, use, and evaluation of specifications.*

The first way is by class of constraint. Different specification languages make it possible to specify different properties of programs. We distinguish two broad classes: those that constrain the behavior of implementations, and those that constrain their structure.

The second classification method is by viewpoint. The behavior of an unobservable program is of no interest. What is considered observable forms the interface of the program. A programming language provides a useful standard definition of what may be observed about programs in that language (e.g., values of nonlocal variables, input/output, exceptional conditions). When the constraints on programs are stated in programming language terms we say the specifications are language-oriented. Other important kinds of constraints may require a viewpoint outside the programming language; for example, the significant behavior may involve external devices whose behavior is controlled or interpreted by humans, and may best be described in terms of abstractions derived from the application domain. We call such specifications application-oriented.

The third classification method is by specification size. Specifications, like programs, come in a great range of sizes. The processes of writing, reading, and checking large specifications differ in important ways from those for small ones. There is no precise boundary between small and large specifications, but when the text of a specification exceeds a few pages, problems of scale begin to dominate.

Three combinations of attributes are so common that we have found it convenient to name them. *System specifications* are application-oriented behavioral specifications of (typically large) collections of programs. They express constraints on a system in terms of what can be observed by its users. *Local specifications* are language-oriented behavioral specifications of single program units. They express constraints on a program in programming language terms, and are typically much smaller than system specifications. Larch interface languages are designed for writing local specifications. *Organizational specifications* combine structural specifications with behavioral specifications of the components. We should be able to demonstrate that an organizational specification implements a system specification by showing that the system will satisfy its specification if each component satisfies its specification.

Our classification is not intended to introduce sharp dichotomies, nor to provide a complete taxonomy. However, it has helped us to focus more carefully in posing and answering a number of key questions about specifications, such as:

- What is accomplished by constructing them?
- What benefits result from their existence?
- When should they be written?
- Who should write them?
- Who should read them?
- Which properties should be used to evaluate them?

The answers can be very different for different kinds of specifications.

*This classification has been abstracted from an article by J.V. Guttag, J.J. Horning, and J.M. Wing "Some Notes on Putting Formal Specifications to Productive Use," *Science of Computer Programming*, Vol. 2, Dec. 1982, pp. 53-68.

The part of the specification following **introduces** declares a set of *operators,* each with its *signature* (the *sorts* of its domain and range). These signatures are used to sort-check *terms* in much the same way function calls are type-checked in programming languages. We use operator, sort, and term in describing the Shared Language to avoid confusion with the similar concepts function, type, and expression in programming languages.

The final part of the specification constrains the operators by means of equations that relate terms containing them. In general, each equation involves several operators, and an operator may appear in several equations.

The first equation resembles a recursive function definition, since the operator eval appears on both the left and right sides. However, it does not fully define eval; it states a relation that must hold among eval, add, and the built-in operator **if then else**. The second and third equations together provide enough information to define the operator ϵ (when applied to any term built up using new and add) in terms of the built-in operators false and $|$, and the operator $=$ for sort Index.

The set of theorems that can be proved about the terms defined in a trait is called its *theory.* It is the infinite set of predicate calculus formulas that consists of the trait's equations, the inequation $\sim$(true $=$ false), and all theorems that can be derived from these formulas plus the axioms and rules of inference of first-order predicate calculus with equality.

The theory associated with TableSpec contains equations and inequations that can be proved by substituting equals for equals. However, there is no metarule stating that if two terms are not provably equal, then they are definitely unequal, nor is there a converse metarule stating that if two terms are not provably unequal, then they are equal. For example, we cannot determine whether add is commutative. The equation

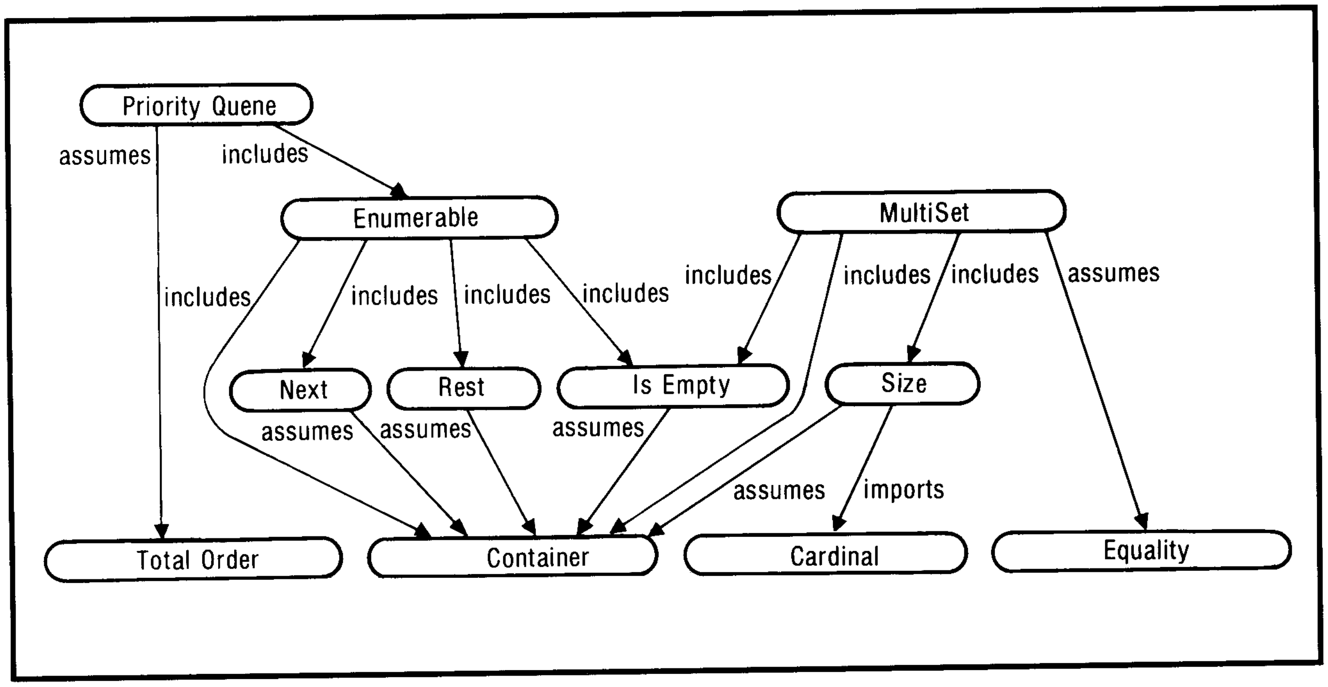


Figure 2. Relations among sample traits.

```
add(add(t, ind, val), indl, val)
    = add(add(t, indl, val), ind, val)
```

is not in TableSpec's theory, but neither is any inequation that would distinguish between the left and right sides. Later, we discuss Larch Shared Language constructs that can be used to generate stronger (larger) theories that contain the answers to such questions.

The next series of examples defines a number of properties that are then combined in different ways to define two traits that correspond to familiar abstract data types. Figure 2 may be used as a road map for these examples, which are presented in a bottom-up fashion, with the exception of the Handbook traits TotalOrder, Cardinal, and Equality, which are used in the examples but not defined here (see specifications box on p. 28 for more details).

The trait Container abstracts the common properties of those data structures that contain elements, such as sets, multisets, queues, and stacks. We have found it useful both as a starting point for specifications of many kinds of containers and as an assumption when defining generic operators.

The new construct in this trait is the **generated by** clause. It indicates that each variable-free term of sort C is equal to some term in which new and insert are the only operators with range C. Thus, it introduces an inductive rule of inference that can be used to prove properties that are true for all terms of sort C.

```
Container: trait
  introduces
    new: → C
    insert: C, E → C
  constrains C so that
    C generated by [ new, insert ]
```

The trait IsEmpty builds on Container by assuming it. It constrains the new and insert operators that it inherits from Container, as well as the operator that it introduces, isEmpty.

The **converts** clause adds nothing to the theory of the trait. It adds checkable redundancy to the specification by indicating that this trait is intended to contain enough axioms to define isEmpty. That is, any variable-free term should be provably equal to one that does not contain isEmpty. Because of the **generated by** inherited from Container, this can be proved by induction over terms of sort C, using new as the basis and using insert(*c*, *e*) in the induction step.

```
IsEmpty: trait
  assumes Container
  introduces isEmpty: C → Bool
  constrains isEmpty, new, insert
    so that for all [ c: C, e: E ]
      isEmpty(new) = true
      isEmpty(insert(c, e)) = false
  implies converts [ isEmpty ]
```

Next and Rest also assume Container. Like **converts, exempts** clauses are concerned with checking, and add nothing to the theory. Here, they indicate that the lack of equations for next(new) and rest(new) is intentional. Even if Next or Rest is included into a trait that claims the convertibility of next or rest, the terms next(new) and rest(new) don't have to be convertible.

```
Next: trait
  assumes Container
  introduces next: C → E
  constrains next, insert so that
    for all [ e: E ]
      next(insert(new, e)) = e
  exempts next(new)

Rest: trait
  assumes Container
  introduces rest: C → C
  constrains rest, insert so that
    for all [ e: E ]
      rest(insert(new, e)) = new
  exempts rest(new)
```

Traits for abstract data types

In a trait that corresponds to an abstract data type (ADT), there will generally be a distinguished sort corresponding to what Guttag[1] calls the type of interest and what Burstall and Goguen[2] call the data sort. In such traits, the operators whose range is the distinguished sort can usually be partitioned into generators, operators that the sort is generated by, and extensions, which can be converted into generators. Operators whose domain includes the distinguished sort and whose range is some other sort are called observers. Observers are usually convertible, and the sort is usually partitioned by one or more subsets of the observers and extensions.

For example, in PriorityQueue (see example in column 1, facing page), the distinguished sort is C, the generators are new and insert, rest is an extension, and the observers are next and isEmpty.

A good heuristic for generating enough equations to adequately define an ADT is to write one for each observer or extension applied to each generator. For PriorityQueue, this rule suggests axioms for rest(new), next(new), isEmpty (new), rest(insert(*q*, *e*)), next(insert(*q*, *e*)), and isEmpty(insert (*q*, *e*)). Note that the trait contains explicit equations for two of the six, and inherits equations for two more from IsEmpty. The remaining two, rest(new) and next(new), are exempted in Rest and Next.

References

1. J.V. Guttag, *The Specification and Application to Programming of Abstract Data Types*, PhD dissertation, Computer Science Department, University of Toronto, Canada, 1975.
2. R. Burstall and J. Goguen, "An Informal Introduction to Specifications Using CLEAR," in *The Correctness Problem in Computer Science*, R. Boyer and J. Moore, eds., Academic Press, New York, 1981, pp. 185-213.

Size assumes Container, and partially defines the size operator. The phrase **imports** Cardinal means that the theory of the importing trait, Size, is a *conservative extension* of the theory of the imported trait, Cardinal. That is, Size's theory contains Cardinal's theory, but does not further constrain any operators appearing in Cardinal, such as 0. Consequently, the operators of Cardinal can be understood independently since they must not be given any new properties in Size.

```
Size: trait
  assumes Container
  imports Cardinal
  introduces size: C → Card
  constrains size so that
    size(new) = 0
```

The Enumerable trait specifies properties common to containers that keep their contents in a definite order, such as stacks, queues, priority queues, sequences, and vectors. It augments Container by combining it with IsEmpty, Next, and Rest. The **includes** clause indicates that Enumerable is intended to inherit their operators and axioms and to further constrain the operators. The assumption of Container by the traits Next, Rest, and IsEmpty is discharged in Enumerable by the explicit inclusion of Container.

The **partitioned by** clause indicates that next, rest, and isEmpty are a complete set of observer functions. That is, if

next($t1$) = next($t2$),
rest($t1$) = rest($t2$), and
isEmpty($t1$) = isEmpty($t2$),
then $t1 = t2$.

```
Enumerable: trait
  includes Container, Next, Rest,
    IsEmpty
  constrains C so that C partitioned by
    [ next, rest, isEmpty ]
```

Building a heritage of specifications

We almost never define new abstractions starting from first principles. The Larch Shared Language examples given here (or any other sequence of simple examples) may give a misleading image of how Larch specifications are developed. Many of the most useful abstractions are already available in *A Larch Shared Language Handbook*, along with many useful building blocks. For example, the traits Container, IsEmpty, Next, Rest, Size, Enumerable, and PriorityQueue are all in the handbook, and would be used "off the shelf" when needed. The handbook trait Bag introduces a number of operators not needed for MultiSet, which causes no problem. However, it is missing the operator numElements. In practice, we would simply include Bag in MultiSet, introduce numElements, and constrain numElements with two equations.

We expect Larch Shared Language traits to be the principal reusable units in Larch. By reusing existing traits, specifiers will save time and avoid errors. Reusing traits drawn from a generally accessible handbook will also serve to standardize notation. We think of handbooks as the concentrated essence of abstractions that experienced specifiers have found useful. The current version contains sections on single-operator properties, binary relations, ordering relations, group theory, numeric types, simple data structures, containers, container operations, nonlinear structures, rings and fields, lattices, enumerated types, and displays. Future versions will contain additional sections.

New traits are unlikely to have as much structure as is present in the various specializations of Container and in other parts of the handbook. This kind of structure tends to come after a large number of related traits have been written and regularities recognized, or when the abstraction represents a well-studied mathematical system. The development of such structure represents a kind of intellectual capital that yields its dividends in future applications.

PriorityQueue specializes Enumerable by further constraining next, rest, and insert. Sufficient axioms are given to convert next and rest. The axioms that convert isEmpty are inherited from the trait Enumerable, which inherited them from the trait IsEmpty.

The **with** clause indicates that the **assumed** trait is TotalOrder with the sort E substituted for the sort T throughout its text.

```
PriorityQueue: trait
  assumes TotalOrder with [ E for T ]
  includes Enumerable
  constrains next, rest, insert so that
    for all [ q: C, e: E ]
      next(insert(q, e)) =
        if isEmpty(q) then e
        else if next(q) ≤ e then next(q)
          else e
      rest(insert(q, e)) =
        if isEmpty(q) then new
        else if next(q) ≤ e
          then insert(rest(q), e) else q
  implies converts [ next, rest, isEmpty ]
```

Finally, MultiSet is a specialization of Container that does not satisfy Enumerable. It combines Container, IsEmpty, and Size, and introduces three new operators.

Constrains MSet is a shorthand for a **constrains** clause listing all operators whose signature includes MSet. The **partitioned by** indicates that count alone is sufficient to distinguish unequal terms of sort MSet. **Converts** [isEmpty, count, delete, numElements, size] is a stronger assertion than the combination of an explicit **converts** [count, delete, numElements, size] with the inherited **converts** [isEmpty].

The **with** clause calls for a substitution of the operator { } for the operator new, as well as the sort MSet for the sort C.

```
MultiSet: trait
  assumes Equality with [ E for T ]
  includes IsEmpty, Size, Container
    with [ MSet for C, { } for new ]
  introduces
    count: MSet, E → Card
    delete: MSet, E → MSet
    numElements: MSet → Card
  constrains MSet so that
    MSet partitioned by [ count ]
    for all [ c: MSet, e, e1, e2 : E ]
      count( { }, e1) = 0
      count(insert(c, e1), e2) =
        count(c, e2) +
          (if e1 = e2  then 1 else 0)
      size(insert(c, e)) = size(c) + 1

      numElements({}) = 0
      numElements(insert(c, e)) =
        numElements(c) +
        (if count(c, e) > 0
          then 0 else 1)

      delete({}), e1) = {}
      delete(insert(c, e1), e2) =
        if e1 = e2 then c else
          insert(delete(c, e2), e1)
  implies converts [ isEmpty, count,
    delete, numElements, size ]
```

The theory associated with any trait includes the theory of each trait that it **assumes, includes,** or **imports.** Thus, Figure 3 is another way of viewing the relations among traits shown in Figure 2.

The theories associated with MultiSet and PriorityQueue say quite a bit about their respective data structures. These structures have much in common, yet also have important differences, such as the order of insertion, which is significant in PriorityQueue but not in MultiSet. Note also some things that have *not* yet been specified about these data structures. We have

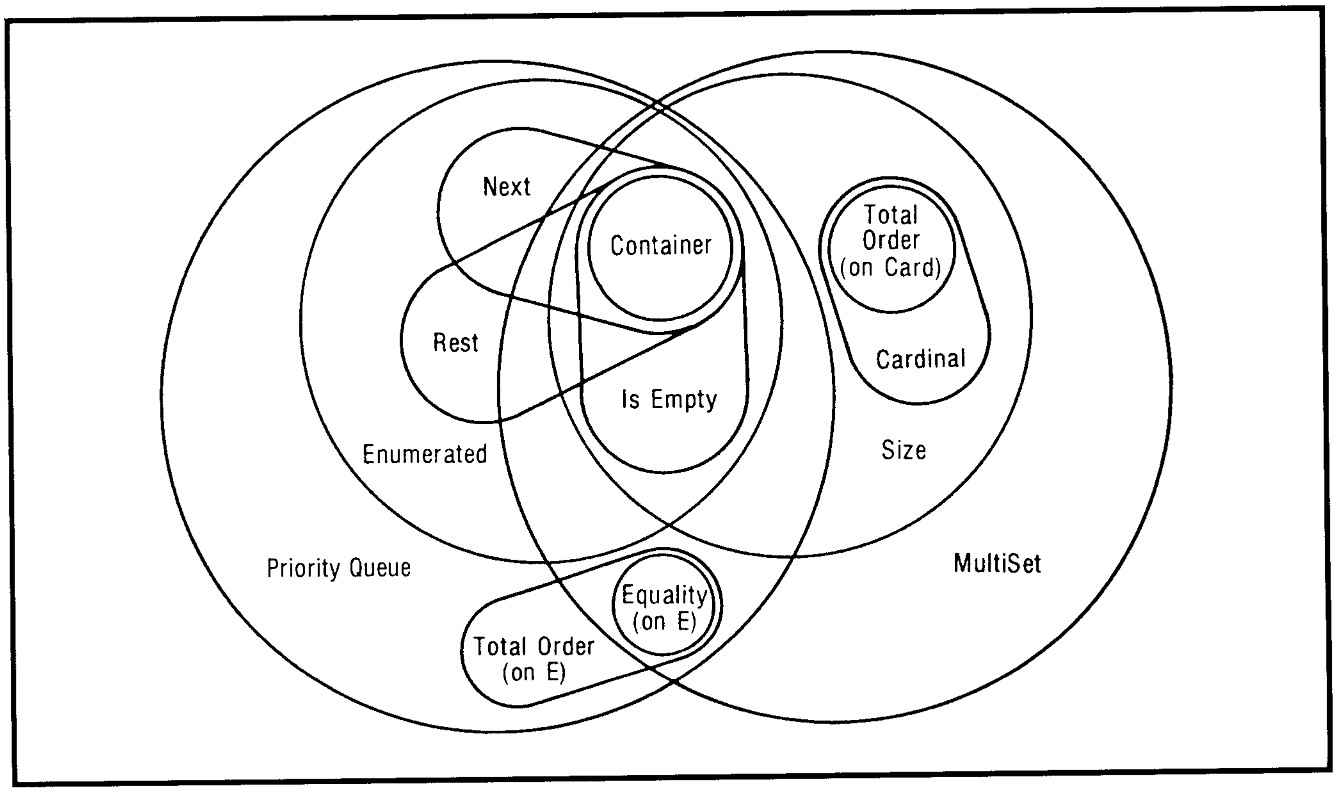


Figure 3. Inclusion relations among the theories of the sample traits.

not specified how they are to be represented. We have not chosen the algorithms to manipulate them. We have not even said what routines are to be provided to operate on them. We have not specified how errors are to be handled. The last two decisions are recorded in interface specifications; the first two are made during implementation.

Larch interface languages

We now turn our attention to interface specifications. It is these specifications that actually describe program units to be implemented. The role of the Larch Shared Language traits is to define the theories that give meaning to operators that appear in the interface specifications.

Each Larch interface language is designed for a particular programming language. Everything from the modularization mechanisms to the choice of reserved words is influenced by the programming language. Larch/Pascal and Larch/CLU are the only two moderately well-developed Larch interface languages to date. A detailed description of the semantics of Larch/Pascal is not yet available, but such a description for an early version of Larch/CLU is given by Wing.[8] A discussion of the style of Pascal programming that Larch/Pascal is designed to support is given by Guttag and Liskov.[9]

We hope to give the flavor of these Larch interface languages with the two small examples that follow. The meaning of programming-language-reserved words is derived directly from their meaning in the programming language. For example, the meaning of **var** in Larch/Pascal is derived from the meaning of **var** in a Pascal parameter list; the meaning of **signals** in Larch/CLU is derived from the meaning of **signals** in CLU.

Both Larch/Pascal and Larch/CLU support the specification of data and procedural abstractions. For each language, we consider one data abstraction, containing several procedural abstractions.

The Larch Project

The Larch Project at MIT's Laboratory for Computer Science and DEC's Systems Research Center is the continuation of collaborative research into the uses of formal specifications that started with the work reported by J. Guttag in 1975 (see bibliography below). It is developing both a family of specification languages and a set of tools to support their use, including language-sensitive editors and semantic checkers based on a powerful theorem prover.

Larch is an effort to test our ideas about making formal specifications useful. To focus the project, we made a number of assumptions, which strongly influenced the directions it has taken.

Local specifications. We started with the belief that behavioral specifications of program units could be useful in the near future. No conceptual breakthroughs or theoretical advances seemed to be needed. Rather, we needed to use what we already knew to design usable languages, develop some software support tools, and educate some system designers and implementers.

Sequential programs. We focused on specifications of the behavior of program units in nonconcurrent environments. We are aware of the importance of concurrency, and of many of the additional problems it introduces. However, we find it hard enough to deal adequately with the sequential case. Furthermore, any useful method for dealing with concurrency must incorporate a way to specify atomic actions.

Scale. Methods that are entirely adequate for one-page specifications may fail utterly for hundred-page specifications. Large specifications must be composed from small ones that can be understood separately, and the task of understanding the ramifications of their combination must be manageable. For large specifications, as Burstall and Goquen have pointed out (see bibliography below), the "putting together" operations are more crucial than the details of the language used for the pieces.

Incompleteness. Realistically, most specifications are going to be partial. Sometimes incompleteness reflects abstraction from details that are irrelevant for a particular purpose; for example, time, storage usage, and functionality might be specified separately. Sometimes it reflects an intentional choice to delay certain design decisions, and sometimes it reflects oversights in design or specification. The checker must be able to detect oversights without rejecting intentional incompleteness.

Errors. Our experience suggests that the process of writing specifications can be as error-prone as the process of programming. We believe that a substantial amount of checking of the specifications themselves must be done. There are two ways to detect errors: human inspection and mechanical checking. We want our specification languages to facilitate the writing of readable specifications. We also want them to incorporate redundancy that will allow mechanical checks to detect the most common errors.

Tools. A serious bar to practical use of formal specifications is the number of tedious and/or error-prone tasks associated with maintaining the consistency of a substantial body of formal text. Tools can assist in managing the sheer bulk of large specifications, in browsing through

(continued on page 31)

(Continued from page 30)
selected pieces, in deriving interactions and consequences, and in teaching a new methodology. Thinking about such tools has changed our ideas about what it is important to include in specification languages. We were strongly influenced by our experience with Affirm (see Musser in bibliography below).

Handbooks. It is inefficient to start each specification from scratch. We need a repository of reusable specification components that have evolved to handle the common cases well, and that can serve as models when faced with uncommon cases. It is no more reasonable to keep reinventing the specifications of priority queues and bit maps than to axiomatize integers and sets every time they are used. The collection should be open-ended, and include application-oriented abstractions, as well as mathematical and implementation-oriented ones.

Language dependencies. The environment in which a program unit 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. Any attempt to disguise this dependence will make specifications more obscure to both the unit's clients and its implementers. On the other hand, many important abstractions in most specifications *can* be defined independently of any programming language.

Bibliography

Burstal, R. and J. Goguen, "Putting Theories Together to Make Specifications," *Proc. Fifth Int'l Joint Conf. Artificial Intelligence*, 1977, pp. 1045-1058.

Forgaard, R., *A Program for Generating and Analyzing Term Rewriting Systems*, SM thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Technical Report MIT/LCS/TR-343, 1985.

Guttag, J.V., *The Specification and Application to Programming of Abstract Data Types*, PhD dissertation, Computer Science Department, University of Toronto, Canada, 1975.

Guttag, J.V., and J.J. Horning, "An Introduction to the Larch Shared Language," *Proc. IFIP Congress 83*, 1983.

Guttag, J.V., and J.J. Horning, *Preliminary Report on the Larch Shared Language*, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., Technical Report MIT/LCS/TR-307, 1983; also issued as Technical Report CSL-83-6, Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, Calif., Dec. 1983.

Guttag, J.V., and J.J. Horning, "A Larch Shared Language Handbook," *Science of Computer Programming, Vol. 6* (to be published).

Guttag, J.V., and J.J. Horning, "Report on the Larch Shared Language," *Science of Computer Programming, Vol. 6* (to be published).

Guttag, J.V., J.J. Horning, and J.M. Wing, *Larch in Five Easy Pieces*, Digital Equipment Corp. Systems Research Center, Report 5, July 1985.

Horning, J.J., "Combining Algebraic and Predicative Specifications in Larch," *Proc. Int'l Conf. Theory and Practice of Software Development* from *Springer-Verlag Lecture Notes in Computer Science*, No. 186, Springer-Verlag. New York, 1985, pp. 12-26.

Kownacki, R., *Semantic Checking of Formal Specifications*, SM thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., 1984.

Musser, D., "Abstract Data Type Specification in the Affirm System," *IEEE Trans. Software Engineering*, Vol. SE-1, 1980, pp. 24-32.

Lescanne, P., "Computer Experiments with the REVE Term Rewriting System Generator," *Proc. 10th ACM Symp. Princ. of Programming Languages*, Jan. 1983, pp. 99-108.

Wing, J.M., *A Two-Tiered Approach to Specifying Programs*, PhD dissertation, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., Technical Report MIT/LCS/TR-299, May 1983.

Wing, J.M., "Helping Specifiers Evaluate Their Specifications," *Proc. Second Software Engineering Conf.*, AFCET, June 1984, pp. 179-191.

Wing, J.M., "Writing Larch Interface Language Specifications," (to be published).

In both Larch interface languages, a specification of a data abstraction (type) has three parts. The first is a header giving the type name and the names of the externally visible routines. The second is an associated trait and a mapping from the types in the data abstraction to sorts in the trait. The third comprises the interface specifications for each routine (procedure or function) of the type. A specification of a routine has three parts: (1) a header giving the name of the routine and the names and types of its formals (parameters and returned values), (2) an associated trait providing the theory of the operators that appear in the body (in the examples, this trait is just the union of the traits associated with the types in the routine's header), and (3) a body stating any requirements on the routine's parameters and specifying the effects the routine must have when those requirements are met.

A sample Larch/Pascal specification. The following Larch/Pascal specification of a data abstraction provides a type, three procedures, and one function:

```
type Bag exports bagInit, bagAdd,
bagRemove, bagChoose
    based on sort MSet from MultiSet
        with [ integer for E ]
    procedure bagInit(var b: Bag)
        modifies at most [ b ]
        ensures b_post = {}
    procedure bagAdd
        (var b: Bag; e: integer)
        requires numElements (insert(b,
            e)) ≤ 100
        modifies at most [ b ]
        ensures b_post = insert(b, e)
    procedure bagRemove
        (var b: Bag; e: integer)
        modifies at most [ b ]
        ensures b_post = delete(b, e)
    function bagChoose
        (b: Bag; var e: integer) : boolean
        modifies at most [ e ]
        ensures
            if ~isEmpty(b)
            then bagChoose &
                count(b, e_post) > 0
            else ~bagChoose &
                modifies nothing
end Bag
```

The body of each routine's specification places constraints on proper arguments for calls on the routine and

How names tie languages together

In an interface specification, we give meaning to names appearing in programs by relating them to names appearing in traits. Thus, it is the names in an interface specification that tie it to traits in the Larch Shared Language and to programs in the programming language. Operators, such as insert; sorts, such as MSet; and trait names, such as MultiSet, provide the link to a theory defined by a collection of traits. Names of routines, such as bagAdd; formal parameters, such as *e;* and types, such as integer, provide the link to programs that implement the specification. It is important not to confuse operators and sorts from the Larch Shared Language with routines and types from the programming language. Operators and sorts appear in specifications, and in reasoning about specifications, but they do not appear in programs. Conversely, routines and types appear in programs but not in traits.

Nondeterminism and incompleteness

Nondeterminism in an interface should not be confused with incompleteness in a trait. We often intentionally introduce operators in traits without giving enough axioms to fully define them (for example, size in Size and new in Container). Sometimes further properties will be given in other traits; sometimes the weaker theory allows greater flexibility in the implementation of an interface. However, it is always the case that for every term t, $t = t$. The whole 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 by the introduction of nondeterministic functions.

Three kinds of induction

Different induction principles can be applied at the Larch Shared Language level, at the interface language level, and at the programming language level. They are all distinct and are useful in proving different kinds of theorems.

Induction over a set of generating operators is used to prove theorems that assert something about all terms of a sort. For example, we might use it to prove by induction over new and insert that hte sum of the counts of all elements in any MSet is equal to its size.

Induction over the specification (often called data type induction) is used to prove something about all legal values of a type. For example, we might show by induction over bagInit, bagAdd, and bagRemove, that no Bag has more than 100 distinct elements. Such a proof would depend on the assumption that objects of type Bag are manipulated only by legal calls on the routines in Bag's specification. Although this restriction is not enforced by Pascal, we could adopt it as a programming convention (see reference 9 at the end of this article).

Induction over an implementation of the type is outside the domain of interface specifications. Rather, it falls under program verification. It can be used, for example, to prove that a representation invariant has been established and preserved.

defines the relevant aspects of the routine's behavior when it is properly called. It can be straightforwardly translated to a predicate over two states in the style of Hehner[10] by combining its three predicates into a single predicate of the form

```
requires predicate =>
  (modifies predicate &
  ensures predicate).
```

An omitted **requires** is interpreted as **true**.

In the body of a Larch/Pascal specification, as in Pascal, the name of a function stands for the value returned by that function. Formal parameters may appear unqualified or qualified by *post.* An unqualified formal stands for the value of that formal when the routine is called. A formal qualified by post, for example, b_{post}, stands for the value of that formal when the routine returns.

The values of variables on entry to and return from routines must be distinguished because Pascal is a language in which statements may alter memory. Since the operators in a Larch Shared Language specification represent functions, this complication does not arise there, nor would it in an interface language for a functional programming language.

The **modifies** predicate is also related to the imperative nature of Pascal. The predicate **modifies at most** $[v_1, ..., v_n]$ asserts that the routine changes the value of no variable in the environment of the caller except possibly some subset of the variables denoted by the elements of $\{v_1, ..., v_l\}$. Notice that this predicate is really an assertion about all variables that do *not* appear in the list, not about those that do. **Modifies at most** is a built-in predicate specific to the programming language. Each Larch interface language comes with its own set of built-in predicates.

The **based on** clause associates the type Bag with the sort MSet that appears in trait MultiSet. This association means that Larch Shared Language terms of sort MSet are used to represent Pascal values of type Bag. For example, the term {} is used to represent the value that *b* is to have when bagInit returns.

The **requires** clause of bagAdd states a precondition that is to be satisfied on each call. It reflects the specifier's concern with how this type can be implemented in Pascal. By putting a bound on the number of distinct elements in the Bag, the specification allows a fixed-size representation. It is quite natural for such considerations to surface in interface specifications; it would not be so natural for them to appear in traits.

The most interesting routine is probably bagChoose. Its specification says that it must set *e* to some value in *b* (if *b* isn't empty), but doesn't say which value. Moreover, it doesn't even require that different invocations of bagChoose with the same value produce the same result; in other words, the implementation may be nondeterministic. Our implementation is abstractly nondeterministic, even though it is a deterministic program (see box at left). The value to which *e* is set depends on the order in which elements have been added to and removed from *b*; whereas this order does not affect *b*'s abstract value.

This interface specification has recorded a number of design decisions beyond those contained in the trait MultiSet. It says which routines must be implemented, and for each routine, it indicates both the condition that must hold at the point of call and the condition that must hold upon return. Thus, a contract that provides a "logical firewall" has been established between the implementers and the clients of type Bag. They can then proceed independently, relying only on the interface specification.

The clients must establish the **requires** clause at each point of call. Having done that, they may presume the truth of the **ensures** clause on return, and that only variables in the **modifies at most** clause are changed. They need not be concerned with how this happens.

The implementers are entitled to presume truth of the **requires** clause on entry. Given that, they must establish the **ensures** clause on return, while respecting the **modifies at most** clause.

Because the interface specification does not specify either the representation of the type or the algorithms in routines, yet another level of design is needed. Because this level is hidden from clients of the data type, the design may be changed without affecting their correctness.

The specification of each routine in an interface can be understood without reference to the specifications of other routines—unlike traits, in which the specification constrains the operators by giving relations among them. Of course, to understand the type itself, to reason about it, or to design an efficient representation for it, the specifications of all its routines must be taken into account.

A sample Larch/CLU specification. Now we use Larch/CLU to specify a bag type. The abstraction is different from the one specified in Larch/Pascal because it exploits features of CLU that do not have analogs in Pascal. However, it is based on the same Larch Shared Language trait.

This example illustrates some of the ways in which programming language dependencies influence interfaces, specifications, and interface languages. Some programming language dependencies are trivial: the syntax has been changed to resemble that of CLU, and routine names don't start with "bag," since in CLU all calls are prefixed with the type name. Some dependencies, however, are more substantial.

```
bag mutable type exports init, add,
 remove, choose

  based on sort MSet from MultiSet
    with [ int for E ]

  init = proc() returns(b: bag)
    modifies nothing
    ensures new(b) & b = {}

  add = proc(b: bag, e: int)
    modifies at most [ b ]
    ensures b_post = insert(b, e)

  remove = proc(b: bag, e: int)
    modifies at most [ b ]
    ensures b_post = delete(b, e)

  choose = proc(b: bag) returns(e: int)
      signals (empty)
    modifies nothing
    ensures
      normally count(b, e) > 0 except
      signals empty when isEmpty(b)
```

In the body of a Larch/CLU specification, an unqualified argument formal stands for the value of the object bound to that argument on entry to the routine. An unqualified result formal stands for the value of the object bound to that argument on exit from the routine.

New is a Larch/CLU built-in predicate. **New**(*b*) in the specification of init asserts that the object bound to

One way to conduct a design review

Formal specifications have many uses. Much research has been done on their application in formal program verification. However, they can also be useful when formal verification is not contemplated. They can, for example, be used to make design reviews more productive.

The following sequence of steps for reviewing the specification of a program unit is adapted from an article by J. Guttag and J. Horning (see reference 6 at the end of this article).

(1) Introduce the unit informally.

(2) Present the language-independent portion of the design: the trait.

(3) Formulate questions (or accusations) about the abstraction defined by the trait.

(4) Examine the trait to answer the questions.

(5) Discuss the suitability of the answers and the virtues of other possible answers.

(6) Present the language-dependent portion of the design: the interface specification.

(7) Formulate questions/accusations about the interface.

(8) Examine the interface specification to answer the questions.

(9) Discuss their suitability and the virtues of other possibilities.

A sample Pascal implementation

To illustrate the relation between an interface specification and an implementation, we give a Pascal implementation of type Bag. Neither the data structure chosen for the representation nor the program itself is very interesting. One of the main goals of data abstraction is to ensure that clients of data types do not need to be interested in their implementations.

Both the abstraction function and the representation invariant are presented informally. If we had included a formal specification of the type used in the representation, we could have presented them formally, using a program annotation language. Then, they could be mechanically combined with the interface specifications already given to derive a concrete specification for each routine, which could then be verified separately.

Notice that the implementation of bagAdd relies on the **requires** clause of its specification.

```
const MaxBagSize = 100;
type
  ElemVals = array [1..MaxBagSize] of integer;
  ElemCounts = array [1..MaxBagSize] of integer;
  Bag = record elems: ElemVals; counts: ElemCounts; end;
{Abstraction function: the abstract bag is equivalent to the result
  of inserting into the empty bag each integer in elems
  a number of times equal to the corresponding number in counts}
{Rep invariant: each integer in counts is at least zero and
  no integer appears in elems more than once associated
  with a positive value in counts}
procedure bagInit(var b: Bag);
  var i: 1..MaxBagSize;
  begin for i := 1 to MaxBagSize do b.counts[i] := 0 end {bagInit};
procedure bagAdd(var b: Bag; e: integer);
  var i, lastEmpty: 1..MaxBagSize;
  begin
    i := 1;
    while (i < MaxBagSize) and (b.elems[i] <> e) do
      begin
        if b.counts[i] = 0 then lastEmpty := i;
        i := i + 1;
      end;
    if b.elems[i] = e
      then b.counts[i] := b.counts[i] + 1
      else begin
        if b.counts[i] <> 0 then lastEmpty := i;
        b.elems[lastEmpty] := e;
        b.counts[lastEmpty] := 1
      end;
  end {bagAdd};
procedure bagRemove(var b: Bag; e: integer);
  var i: 1..MaxBagSize;
  begin
    i := 1;
      while ( not((b.elems[i] = e) and (b.counts[i] > 0)) and
            i < MaxBagSize ) do
            i := i + 1;
      if (b.elems[i] = e) and (b.counts[i] > 0) then
        b.counts[i] := b.counts[i] − 1
  end {bagRemove};
function bagChoose(b: Bag; var e: integer): boolean;
  var i: 1..MaxBagSize;
  begin
    i := 1;
      while (i < MaxBagSize) and (b.counts[i] = 0) do i := i + 1;
      if b.counts[i] = 0
        then bagChoose := false {e not modified}
        else begin e := b.elems[i]; bagChoose := true end
  end {bagChoose};
```

b when the routine returns is distinct from all previously accessible objects. This forbids init to return an alias for an existing bag. Larch/Pascal has a built-in predicate with a similar meaning, but it is used less often because fewer Pascal interfaces deal with dynamically allocated variables.

The built-in types of CLU, unlike those of Pascal, offer no incentive to place an a priori bound on the size of objects. Thus, there is no **requires** clause in the specification of add.

The use of **signals** is another CLU-specific aspect of the specification. The CLU choose has a rather different header than does the Pascal bagChoose. CLU interfaces are typically designed to use CLU's exception-handling mechanism rather than returning flag values. To make it easy to specify permitted and required signals, Larch/CLU contains some special syntactic sugar. A predicate of the form

```
normally Normal Predicate except
signals Signal Name
  when Exception Guard
```

is a shorthand for the predicate

```
(returns | signals Signal Name) &
(returns => (~ Exception Guard &
  Normal Predicate)) &
(signals Signal Name =>
  Exception Guard)
```

where **returns** and **signals** are Larch/CLU built-in predicates that deal with the possible ways for routines to terminate.

Notes on two-tiered specifications

Larch can be used to write specifications that resemble operational specifications built on abstract models.[11,12] The Larch approach, however, differs in several important respects. The Larch Shared Language is used to specify a theory, rather than a model, and the Larch interface languages are built around predicate calculus rather than around an operational notation. One consequence of these differences is that Larch specifications are less prone to implementation bias.

It would be complicated to give semantic definitions of Larch/Pascal and Larch/CLU directly, because Pascal and CLU are complicated. Instead, we define the interface language semantics relative to the programming language semantics. This approach has two main advantages: we can be quite precise about what it means for an implementation to satisfy a specification, and we can provide a straightforward translation of a Larch interface language into predicate calculus.

The Larch Shared Language has mechanisms for building one specification from another (**assumes, includes,** and **imports**), and for inserting checkable redundancy into specifications (**constrains** and **converts**). The Larch interface languages do not have corresponding mechanisms. We wish to encourage a style of specification in which most of the programming-language-independent complexity is pushed into the traits, allowing interface specifications to become almost trivial. We feel that specifiers are less likely to make serious mistakes in the simpler domain. Furthermore, it should be easier to provide machine support to help them catch the mistakes they do make. Finally, by encouraging specifiers to concentrate their efforts on the traits, we increase the likelihood that parts of specifications will be reusable—not only for different specifications written in the same Larch interface language, but also across specifications written in different Larch interface languages.

The semantics of the Larch Shared Language are quite simple—except for some of the static error checking. This simplicity stems primarily from two decisions:

(1) All operators and sorts appearing in shared specifications are treated as "auxiliary"; that is, operators and sorts need never be implemented.

(2) Issues are not dealt with in the Larch Shared Language if they must also be dealt with at the Larch interface language level.

As a result of the first decision, there is no mechanism to support the hiding of operators in the Larch Shared Language. The hiding mechanisms of other specification languages allow the introduction of auxiliary operators that don't have to be implemented. These operators are not completely hidden, since they must be read to understand the specification, and they are likely to appear in reasoning based on the specification. Since none of the operators appearing in a Larch Shared Language specification are to be implemented, the introduction of a hiding mechanism would have no effect.

As a result of the second decision, there is no mechanism other than sort checking for restricting the domain of operators. Terms such as eval(new, *i*) in TableSpec are considered well-formed. Furthermore, no special error elements are introduced to represent the values of such terms. All preconditions and errors are handled at the Larch interface language level. The Larch Shared Language does include a mechanism for indicating that meanings of certain terms, such as eval (new, *i*), have been intentionally left unconstrained. It may be desirable to check that the meaning of an interface specification does not depend on the meaning of **exempt** terms.

In this article we have discussed the Larch Shared Language before the Larch interface languages. This does not mean that traits are always written before the interface specifications that are based on them. In practice, we usually start by writing a trait, but we often go back and amend traits as we write interface specifications. In particular, we frequently add operators that enable us to write our predicates more concisely.

The ideas behind the Larch Project are more important than its details—although a large number of details must be gotten right before the pieces can fit together. A useful method is more than a collection of separately good ideas.

It is too soon to draw conclusions about the utility of Larch in software development. We have written a significant number of Larch Shared Language specifications. On the whole, we were pleased with the specifications, and with the ease of constructing them. Some relatively primitive tools uncovered many errors for us. We uncovered some more subtle design errors by inspection; we are encouraged by the fact that many of these errors would have been uncovered by (as yet unimplemented) checks called for in the language definition. However, until we have completed better tools that allow us to gain some experience with automated semantic checking, we cannot know just how helpful these checks will be.

We have not yet written many specifications in Larch interface languages. The experience we have had, however, leaves us optimistic. In particular, we have been pleased with the Larch style of two-tiered specification. We are presently in the process of designing and documenting some Larch interface languages and plan to begin writing some specifications in them. That experience should give us a much firmer basis for evaluating the Larch Shared Language, Larch interface languages, and the Larch style of specification. □

Acknowledgments

We thank Butler Lampson and Soren Prehn for helping us to improve the exposition of our ideas. Leslie Lamport responded graciously and helpfully to a request for a very fast review of the penultimate version.

Several members of the Larch Project have contributed ideas, criticism, and trial implementations to the development of Larch. Randy Forgaard, Ron Kownacki, and Joe Zachary deserve special thanks.

IFIP Working Group 2.3 (Programming Methodology) provided both a continuing education and a constructively critical sounding board.

Jean-Raymond Abrial, Rod Burstall, Susan Gerhart, Cliff Jones, Barbara Liskov, Bill McKeeman, Dave Musser, Doug Ross, Mary Shaw, Jim Thatcher, and Steve Zilles were all, in their own ways, significant influences on this work.

The Larch Project and related research were supported at MIT's Laboratory for Computer Science by DARPA under contract N00014-75-C-0661 and by the National Science Foundation under grant MCS-811984-6, at the Digital Equipment

Corp. Systems Research Center and at the Xerox Palo Alto Research Center by corporate funds, and at USC by the National Science Foundation under grant ECS-8403905.

References

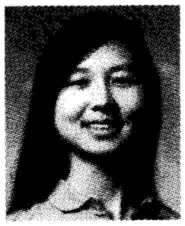
1. B. Meyer, "On Formalism in Specifications," *IEEE Software*, Vol. 2, No. 1, Jan. 1985, pp. 6-26.
2. *Proc. Int'l Joint Conf. Theory and Practice of Software Development*, Vols. 1 and 2 from *Springer-Verlag Lecture Notes in Computer Science*, Nos. 185 and 186, Springer-Verlag, New York, 1985.
3. D. Sannella and A. Tarlecki, "Some Thoughts on Algebraic Specification" (to be published).
4. L. Lamport, "What It Means for a Concurrent Program To Satisfy a Specification: Why No One Has Specified Priority," *Proc. 12th ACM Symp. Princ. of Programming Languages*, Jan. 1985, pp. 78-83.
5. M. Shaw, "Abstraction Techniques in Modern Programming Languages," *IEEE Software*, Vol. 1, No. 4, Oct. 1984, pp. 10-26.
6. J. Guttag and J. Horning, "Formal Specification as a Design Tool," *Proc. ACM Symp. Princ. of Programming Languages*, Jan. 1980, pp. 251-261.
7. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics: EATCS Monographs on Theoretical Computer Science*, Vol. 6, Springer-Verlag, New York, 1985.
8. J. Wing, *A Two-Tiered Approach to Specifying Programs*, PhD dissertation, Technical Report MIT/LCS/TR-299, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., May 1983.
9. J. Guttag and B. Liskov, *Abstraction and Specification in Program Design*, MIT Press/McGraw Hill (to be published).
10. E. Hehner, "Predicative Programming, Parts I and II," *Comm. ACM*, Vol. 27, Feb. 1984, pp. 134-151.
11. C. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica*, Vol. 1, 1972, pp. 271-281.
12. V. Berzins, *Abstract Model Specifications for Data Abstractions*, Technical Report MIT/LCS/TR-221, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., July 1979.

John V. Guttag is an associate professor in MIT's Department of Electrical Engineering and Computer Science and a member of MIT's Laboratory for Computer Science. His current research interests include the application of formal specifications to programming and the theory and application of term-rewriting systems. He was previously on the faculty of the University of Southern California. He received an AB and MS from Brown University and a PhD from the University of Toronto.

His address is MIT Laboratory for Computer Science, 545 Technology Sq., Cambridge, MA 02139.

James J. Horning is a member of Digital Equipment Corp.'s Systems Research Center. His long-term technical interest is the mastery of complexity in computer systems. He was previously a member of the Computer Science Laboratory at the Xerox Palo Alto Research Center and the Computer Systems Research Group at the University of Toronto. He has served as Chairman of IFIP Working Group 2.3 (Programming Methodology) and programming languages editor of *Communications of the ACM*. He received a PhD from Stanford, an MS from UCLA, and a BA from Pacific Union College.

His address is Digital Equipment Corporation, 130 Lytton Ave., Palo Alto, CA 94301.



Jeannette M. Wing is an assistant professor in Carnegie-Mellon University's Department of Computer Science. Her current research interests include the application of formal specifications to large, complex systems, and language design for geometric reasoning. She was previously on the faculty of the University of Southern California. She received an SB, an SM and a PhD from MIT.

Her address is Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.