COMBINING ALGEBRAIC AND PREDICATIVE SPECIFICATIONS IN LARCH

J. J. Horning
Systems Research Center
Digital Equipment Corporation
130 Lytton Avenue
Palo Alto, CA 94301 U. S. A.

Abstract

Recently there has been a great deal of theoretical interest in formal specifications. However, there has not been a corresponding increase in their use for software development. Meanwhile, there has been significant convergence among formal specification methods intended for practical use.

The Larch Project is developing tools and techniques intended to aid in the productive use of formal specifications. This talk presents the combination of ideas, both old and new, that we are currently exploring.

One reason why our previous specification methods were not very successful was that we tried to make a single language serve too many purposes. To focus the Larch Project, we made some fairly strong assumptions about the problem we were addressing.

Each Larch specification has two parts, written in different languages. Larch interface languages are used to specify program units (e.g., procedures, modules, types). Their semantics is given by translation to predicate calculus. Abstractions appearing in interface specifications are themselves specified algebraically, using the Larch Shared Language.

A series of examples will be used to illustrate the use of the Larch Shared Language and the Larch/CLU interface language. The talk will conclude with notes on the key design choices for each of the languages, and for the method of combining the two parts of a specification.

Introduction

I would like to begin with three general observations about the field of formal specification:

Theoretical interest: It is clear that formal specification has captured the interest of a signficant group in the theoretical computer science community. The program for this joint conference gives evidence of the vitality of the research area. Similar evidence will be found in a dozen other recent conferences and in numerous journals. Sound theoretical foundations are being given for more and more different kinds of formal specification languages, and subtle semantic problems are being explored to ever-greater depth, particularly in such areas as parameterization and concurrency.

Adoption: It is equally clear that this interest has not been matched by the use of formal specifications in software development. Several attempts—and even some successes—have been reported. But formal methods have not swept the programming community (at my employer, or in the world at large) in the same way that higher-level languages did in the decade after FORTRAN. In retrospect, perhaps this should have been expected, for a variety of reasons. Many of the theoretical results are not presented in a form accessible to practitioners. It is not obvious how some of the theoretically appealing methods deal with many problems of practical importance. Formality is not inevitable in specifications—there are more alternatives for human/human communication than for human/machine communication. Few of the formal specification languages have come with the quality of computer support that even the original FORTRAN did.

Convergence: As developers of formal specification methods confront the problems of practical software development, many distinctions that formerly seemed clear-cut are becoming blurred. Everyone seeks to combine the advantages traditionally associated with each method, while mitigating its disadvantages. Good ideas are borrowed freely and hybrid methods are tried. It is difficult to track "intellectual ancestry," and generalizations about the limitations of classes of methods are quickly falsified.

In this talk, I would like to share the particular combination of old and new ideas that we are currently exploring in the Larch Project. We do not yet have much experience with their use in practical software development, and the supporting tools are not yet available. But we are pleased with the way the pieces seem to be fitting together. We hope that software developers will assess their promise for dealing with practical problems, and that developers of other specification methods will consider including some of them in their own schemes.

Context: The Larch Project

The Larch Project at MIT's Laboratory for Computer Science and DEC's Systems Research Center is the continuation of more than a decade of collaborative research with John Guttag and his students in the area of formal specification. 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 [Lescanne 83][Forgaard 84].

Larch is an effort to test our ideas about making formal specifications useful. We tried to analyze the reasons why our previous specification methods had not been as useful (and hence not as widely used) as we had hoped they would be [Guttag, Horning, and Wing 82]. We identified several problems to be solved before we could confidently offer our methodology to software developers. To focus the project, we made the following assumptions, which strongly influenced the directions it has taken:

Local specifications: We started with the belief that programming-language-oriented 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 focussed on specifications of the behavior of program units in non-concurrent environments. We are aware of the importance of concurrency, and of many of the additional problems it introduces. However, we find it quite hard enough to deal adequately with the sequential case. A successful framework for dealing with concurrency will still need a method for specifying the atomic actions of the concurrent system.

Scale: Methods that are entirely adequate for one-page specifications may fail utterly for hundred-page specifications. It is essential that large specifications be composed from small ones that can be understood separately, and that the task of understanding the ramifications of their combination be managable. For large specifications, the "putting together" operations [Burstall and Goguen 77] are more crucial than the details of the language used for the pieces of which it is composed.

Incompleteness: Realistically, most specifications are going to be partial. Sometimes incompleteness reflects abstraction from details that are irrelevant for a particular purpose; e.g., 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 the design or specification process. It is important to detect the latter kind of incompleteness without making the other two kinds awkward.

Errors: Our experience suggests that the process of writing specifications is at least as error-prone as the process of programming. We believe that it is important to do a substantial amount of checking of the specifications themselves. The ultimate tool for error-detection is the understanding of human minds. However, we have found that—by designing the specification language to incorporate useful redundancy—some surprisingly effective mechanical checks are feasible. We have chosen to supplement checking analogous to a compiler's syntax and type checking with a number of semantic checks that rely on a theorem prover.

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 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.

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 bitmaps than to axiomatize integers and sets every time they are used. A rich collection of "abstract models" is a step in the right direction, but the collection should be open-ended, and include application-oriented abstractions, as well as mathematical and implementation-oriented ones. We expect the most useful parts of specification handbooks to be written in the Shared Language.

Language dependencies: For many years we tried to write specifications in languages that were completely free of bias towards any programming language. We now think that effort was misdirected, at least for local specifications. 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 users and its implementers. On the other hand, many of the important abstractions in most specifications *can* be defined in a language-independent way.

We have adopted a two-tiered methodology [Wing 83]. Each specification has two parts, written in different languages: specifications of program units are written in a Larch interface language that is tailored to a programming language; these specifications use programming-language-independent abstractions, which are specified separately in the Larch Shared Language. Some important aspects of the Larch family of specification languages are:

Composability: The Larch Shared Language is oriented towards the incremental construction of specifications from other specifications.

Emphasis on presentation: To make it easier to read and understand specifications, the composition mechanisms in the Larch Shared Language are defined as operations on specifications, rather than on theories or models.

Interactive and integrated with tools: The Larch languages are intended for interactive use. Tools are being constructed for both interactive construction and incremental checking of specifications.

Semantic checking: The semantic checks for the Larch languages were designed assuming the availability of a powerful theorem prover. Hence they are more comprehensive than the syntactic checks commonly defined for specification languages.

Shared Language based on equations: The Larch Shared Language has a simple semantic basis taken from algebra. However, because of the emphasis on composability, checkability, and interaction, it differs substantially from the algebraic specification languages we have used in the past.

Interface languages based on predicate calculus: Each interface language is a way to write assertions about states, that can be translated to formulas in typed first-order predicate calculus with equality. Programming-language-specific notations deal with constructs such as side effects, exception handling, and iterators. Equality over terms is defined in the Shared Language; this provides the link between the two parts of a specification.

Example Specifications in the Larch Shared Language

The following series of examples is intended to give the flavor of the Larch Shared Language. A complete description of the Larch Shared Language is contained in [Guttag and Horning 85a], and extensive examples of its use are given in [Guttag and Horning 85b].

The *trait* is the basic module of specification. A trait may specify an abstract data type, but frequently traits are used to capture general properties that may be shared by many types. Such traits may be included in other traits that specify particular types.

Consider the following specification describing tables that store values in indexed places:

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 [*ind1, ind2:* Index, *val:* Val, *t:* Table]
eval(add(*t, ind1, val*), *ind2*) = **if** *ind1* = *ind2* **then** *val* **else** eval(*t, ind2*)
ind1 ∈ new = false
ind1 ∈ add(*t, ind2, val*) = (*ind1* = *ind2*) ∨ (*ind1* ∈ *t*)
size(new) = 0
size(add(*t, ind1, val*)) = **if** *ind1* ∈ *t* **then** size(*t*) **else** size(*t*) + 1
isEmpty(*t*) = (size(*t*) = 0)

This example is similar to a conventional algebraic specification in the style of [Guttag and Horning 80]. The part of the specification following **introduces** declares a set of *operators* (function identifiers), each 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. The remainder of the specification constrains the operators by writing equations that relate sort-correct terms containing them.

Each trait defines a a set of well-formed formulas (wff's) of predicate calculus that is closed under inference. The theory associated with a simple trait written in the Larch Shared Language is defined by:

Axioms: Each equation, universally quantified by the variable declarations of the containing **constrains** clause, is in the theory.

Inequation: ¬ (true = false) is in the theory. All other inequations in the theory are derivable from this one and the meaning of equality.

Predicate calculus: The theory contains the conventional axioms of typed first-order predicate calculus with equality, and is closed under its rules of inference.

The next example is an abstraction of those data structures that "contain" elements, e.g., set, bag, queue, stack. It is useful both as a starting point for specifications of various kinds of containers, and as an assumption for generic operators. The crucial part of the trait is the **generated by**. By indicating that any term of sort C is equal to some term in which new and insert are the only operators with range C, it introduces an inductive rule of inference that can be used to prove properties of terms of sort C.

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

The next example builds upon Container by assuming it. It constrains the new and insert operators it inherits from Container, as well as the operator it introduces, isEmpty. The **converts** clause adds nothing to the theory of the trait. It adds checkable redundancy by indicating that this trait is intended to contain enough axioms to adequately specify isEmpty. Because of the **generated by**, this can be proved by induction over terms of sort C, using new as the basis and 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]

The next two examples also assume Container. Like **converts**, the **exempts** clauses are concerned with checking, and add nothing to the theory. 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 converted.

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)

The next example specifies properties common to various data structures such as stacks, queues, priority queues, sequences, and vectors. It augments Container by combining it with IsEmpty, Next, and Rest. **Includes** indicates that this trait is intended to inherit their operators, and to constrain them further. Specifically, the **partitioned by** clause constrains new, insert, and isEmpty by indicating that they are a "complete" set of observer functions. I.e., if two terms are different, this difference can be observed in the value of at least one of these functions. Since little other information has been supplied about these operators, the **partitioned by** does not yet add much to the associated theory.

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

The next example 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.

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) $\leq$ e **then** next(q) **else** e
rest(insert(q, e)) = **if** isEmpty(q) **then** new
else if next(q) $\leq$ e **then** insert(rest(q), e) **else** q
implies converts [next, rest, isEmpty]

The next example illustrates a specialization of Container that does not satisfy Enumerable. It augments Container by combining it with IsEmpty and Cardinal, and introducing two new operators. Container and IsEmpty are **included** because the trait further constrains operators inherited from them. Cardinal is **imported**; this indicates that this trait inherits Cardinal's specification, but is not intended to further constrain any of its operators. The theory associated with Cardinal can thus be understood independently, and will not be enriched by MultiSet. **Imports** and **includes** yield the same theory, but stronger checks are performed for **imports**.

The **partitioned by** indicates that count alone is sufficient to distinguish unequal terms of sort MSet. **Converts** [isEmpty, count, delete] is a stronger assertion than the combination of an explicit **converts** [count, delete] with the inherited **converts** [isEmpty].

MultiSet: **trait**
assumes Equality **with** [E **for** T]
imports Cardinal
includes Container **with** [MSet **for** C, {} **for** new], IsEmpty **with** [MSet **for** C]
introduces count: E, MSet $\rightarrow$ Bool
delete: E, MSet $\rightarrow$ MSet
size: MSet $\rightarrow$ Card

constrains MSet **so that**
MSet **partitioned by** [count]
for all [*c:* MSet, *e1, e2:* E]
count({}, *e1*) = 0
count(insert(*c*, *e1*), *e2*) = count(*c*, *e2*) + (**if** (*e1* = *e2*) **then** 1 **else** 0)
size({}) = 0
size(insert(*c*, *e1*)) = size(*c*) + 1
delete({}, *e1*) = {}
delete(insert(*c*, *e1*), *e2*) = **if** *e1* = *e2* **then** *c* **else** insert(delete(*c*, *e2*), *e1*)
implies converts [isEmpty, count, delete]

An Example Specification in Larch/CLU

Theories are all very well, but what is their connection to software development? In Larch, the theories associated with specifications written in the Shared Language are used to give meaning to operators appearing in specifications written in Larch interface languages. It is these interface specifications that actually provide information about program units.

Interface languages are programming-language dependent. Everything from the modularization mechanisms to the choice of reserved words is influenced by the programming language. At present, there is only one moderately well-developed Larch interface language, the Larch/CLU language [Wing 83][Guttag, Horning, and Wing 85]. The semantics of Larch/CLU incorporates semantic constructs from CLU. For example, the meaning of **signal** in Larch/CLU derives from the meaning of **signal** in CLU—which is different from the meaning of SIGNAL in PL/I or MESA. Correspondingly, Larch/CLU uses CLU-like syntax for constructs in common, e.g., procedure headers. Other interface languages would use concepts and terminology based on their programming languages.

I will present just one short specification written in a version of Larch/CLU to give the flavor of the language. The specification defines a type, ten_bag, together with four procedures. It would be implemented by a CLU *cluster*.

ten_bag **mutable type exports** singleton, add, remove, choose

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

singleton = **proc**(*e:* int) **returns**(*b:* ten_bag)
modifies nothing
ensures new(*b*) ∧ *b* = insert({}, *e*)

add = **proc**(*b:* ten_bag, *e:* int) **signals** (too_big)
modifies at most [*b*]
ensures normally b_{post} = insert(b_{pre}, *e*)
except signals too_big **when** size(b_{pre}) = 10
ensuring modifies nothing

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

    choose = proc(b: ten_bag) returns(e: int)
        requires ¬ isEmpty(b)
        modifies nothing
        ensures count(b, e)  0

end ten_bag
```

The specification of each procedure can be straightforwardly translated to a predicate over two states in the style of [Hehner 84]. The **requires** clause, if present, represents a precondition that may be assumed by the implementation, and must be ensured by any caller. The **modifies** clause places a bound on the objects the procedure is allowed to change. The **ensures** clause is like a postcondition, but may reference values of objects in both the pre and post states.

The names in a Larch/CLU specification tie it to two other kinds of formal text: traits in the Shared Language, and programs in CLU. Operators (e.g., insert), sort names (e.g., MSet), and trait names (e.g., MultiSet) provide the link to a theory defined by a collection of traits. Names of procedures (e.g., add), formal parameters (e.g., *e*), types (e.g., int), and signals (e.g., too_big) provide the link to programs that implement the specification. The primary job of an interface language is to bring these two together. For example, the **based on** clause connects type names and sort names. The **requires** and **ensures** clauses contain operators, formal parameters, and signal names. These are used together to constrain the relationship between the values of the actuals on entry to a procedure and their values on exit from the procedure.

Each procedure's specification can be studied in isolation—in contrast to traits, where the core of the specification involves the interactions among operators. Of course, to understand or reason about the type, it is still necessary to consider the specifications of all its procedures. CLU's type-checking ensures the soundness of a data type induction principle for this type. This would enable us to prove that the size of any ten_bag value generated by a non-erroneous program is less than or equal to 10.

Induction over the procedures of a data type is distinct from induction over the generating operators of a sort, and is used to prove theorems about values in a different space. Each value of type ten_bag can be represented by a term of sort MSet, but not every term represents a value that can be obtained using the procedures of the type. For example, induction over {} and insert can be used to prove that for every term of sort MSet there is term representing a larger MSet. This is not in conflict with the proof mentioned in the previous paragraph.

Choose is probably the most interesting procedure in this example. Its specification says that it must return some value in the ten_bag it is passed, but doesn't say which value. Moreover, it doesn't even require that different invocations of choose with the same argument produce the same result. Choose is an example of nondeterminism, and therefore cannot be specified by equating its result to a term.

This specification can be satisfied by a CLU cluster implementing one type, ten_bag, with four procedures, singleton, add, remove, and choose. The specification says nothing about "implementing" sorts (such as MSet) or operators (such as {} and insert). These auxiliary constructs are defined solely for the purpose of writing interface specifications; they do not exist in programs.

Execution errors, on the other hand, are properties of programs; they do not exist in traits. **Requires** clauses and **signals** provide means to specify two different ways of dealing with erroneous conditions. For example, add must raise a signal if adding another element would make its argument too big, whereas the implementor of choose is allowed to assume that it will not be called with an empty ten_bag. Note that this precondition cannot be checked at runtime by a program using the type ten_bag. The size operator is available for reasoning about procedures, but the interface (as specified) does not supply any corresponding procedure. Ensuring that execution of a program using this type is error-free would thus require some sort of (formal or informal) proof about the program.

Notes on the Larch Shared Language

Why Algebra? This question really expands into two questions: Why not an operational or abstract model approach? and Why not full predicate calculus? The short answer is that we find equations to be a convenient notation for stating properties of many abstractions useful in programming. A more detailed answer would cover such issues as the ease with which partial specifications can be combined to yield stronger specifications, the ease with which partial specifications can be read and understood, the descriptive power of the technique, and the suitability of the formalism for efficient theorem-proving using rewrite rules.

The Shared Language is perhaps more notable for what it leaves out than for what it includes. We tried to keep it simple. However, before omitting a feature found in other algebraic specification languages, we had to convince ourselves that the gain in simplicity of the language was worth the cost in expressive power or increase in the complexity of specifications written in the language.

A key assumption underlying our design was that specifications should be constructed and reasoned about incrementally. This led us to a design that ensures that adding things to a trait never removes formulas from its associated theory. The desire to maintain this monotonicity property led us to construe the equations of a trait as denoting a first-order theory. Had we chosen to take the theory associated with either the initial or final interpretation of a set of equations (as in [ADJ 78] and [Wand 79]), the monotonicity property would have been lost.

In a trait that defines an "abstract data type" there will generally be a distinguished sort corresponding to the "type of interest" of [Guttag 75] or "data sort" of [Burstall and Goguen 81]. In such traits, it is usually possible to partition the operators whose range is the distinguished sort into "generators," those operators which 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.

While we expected that many traits would correspond to complete abstract data types, we expected that even more would not. This led us to introduce **generated by** and **partitioned by** as independent constructs. **Generated by** is used to close the set of constructors of a sort, and **partitioned by** to indicate that a set of observers is complete. Separating these constructs affords the specifier some useful flexibility.

The ability to substitute for any operator or sort identifier appearing in a trait, using a **with** list, is very powerful. In effect, all such identifiers are formal parameters. An earlier version of the Larch Shared Language had explicit lambda abstraction in traits. However, we discovered that our assumptions at the time when a trait was written about which names should be parameters too often limited their applicability. We often wished to substitute for a name that had not been listed as a parameter. Even more often, we found ourselves using the same identifier for the actual as the formal, because most of the potential parameterization was not needed for a specific use.

It is the trait's text that is effectively parameterized by sort and operator identifiers, rather than the associated theory. This allows us to completely sidestep the subtle semantic problems associated with parameterized theories, theory-valued parameters, and the like—some of which are dealt with in other presentations at this conference.

We have chosen not to use "higher-order" entities in defining the Larch Shared Language. Traits are simple textual objects, combined by operations defined on their text. We have found that such operations are much easier to explain to readers of specifications than are operations on theories or models. Of course, for each of our combining operations on traits, there is a corresponding operation on theories such that the theory associated with a combination is equivalent to the combination of the associated theories, so the difference is largely one of exposition.

Notes on Interface Languages

Why predicates on two states? It should scarcely be necessary to justify the use of predicate calculus in program specification, since it crops up in so much work dealing with precise descriptions of programs—starting from Turing and von Neumann, and running through Floyd, Hoare, Dijkstra, and many more—uses this notation. But we have followed Hehner and Jones in using predicates over two states, rather than one. This seems to work out well both as a tool for describing the semantics of programming languages and as a tool for stating requirements on particular programs. In practice, we have found our specifications easier to write and to read since we adopted the two-state notation, but I know of no formal justification for this.

Notes on Combining an Algebraic and a Predicative Language

This section touches on some of the more important ramifications of the way Larch Shared Language and Larch interface language specifications fit together.

The style of specification used in Larch resembles that used in operational specifications built upon abstract models. It differs, however, in several important respects. The Shared Language

is used to specify a theory rather than a model, and the interface languages are built around predicate calculus rather than around an operational notation. One consequence of these differences is that Larch specifications never exhibit "implementation bias."

The semantic bases of Larch interface languages tend to mirror the semantic bases of the programming languages from which they are derived. In general, this means that the semantics of an interface language is rather complex. But it does allow us to be quite precise about what it means for an implementation to "satisfy" a Larch specification.

The semantics of the Larch Shared Language is quite simple, largely as a consequence of two decisions:

Operators and sorts appearing in traits are auxiliary and are not part of the implementation.

Issues that must be dealt with at the interface language level are not tackled again at the Shared Language level.

As a result of the first decision, there is no mechanism to support the hiding of operators in the 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 are likely to appear in reasoning based on the specification. Since none of the operators appearing in a Shared Language specification is intended to be implemented, the introduction of a hiding mechanism could have no effect on the set of implementations satisfying a Larch specification.

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*) are well-formed, even though there are no equations that allow them to be simplified. Furthermore, no special "error" elements are introduced to represent the "values" of such terms. Preconditions and errors are handled at the interface language level.

Similarly, nondeterminism is left to the interface language. Nondeterminism in an interface should not be confused with incomplete specification in a trait. We often intentionally introduce operators in traits without giving enough axioms to fully define them. That is to say, there are distinct terms that are neither provably equal nor provably unequal. 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." It is also not generally true for "functions" in most programming languages.

Issues of name scoping are also left to the interface language level. The Larch Shared Language does not "qualify" operator or sort names with the traits in which they are introduced or defined. Thus, within a trait, all such names (including those acquired from other traits) are "global." This is extremely helpful when combining a number of traits to specify a single type, but raises the possibility of accidental "collisions." Although we do not have a lot of experience yet, we expect two features of the language to keep this from becoming a serious problem: an operator's signature is treated as part of its name, so that two operators with different signatures can never

collide; and imports checking ensures that a trait does not add new constraints to operators being acquired.

A Larch Shared Language trait does not have block structure, and there is no "hierarchy" in its associated theory. We do not expect this to be a problem, because the traits needed to specify single program units should be relatively small.

While the semantic basis of Larch/CLU is considerably more complicated than that of the Larch Shared Language, its static semantics is considerably simpler. In the Shared Language, there are several mechanisms for building a specification from other specifications and for inserting checkable redundancy into specifications. Corresponding mechanisms are not present in Larch/CLU. Interfaces are specified in terms of traits, not in terms of other interfaces.

We wish to encourage a style of specification in which most of the structural complexity is pushed into the Shared Language part of specification. We feel that specifiers are less likely to make serious mistakes in this simpler domain. Furthermore, it should be easier to provide machine support that will help specifiers to catch the mistakes that they do make. Finally, by encouraging specifiers to put effort into Shared Language specifications, we increase the likelihood that parts of specifications will be reusable.

Concluding Remarks

The ideas behind the Larch Project are more important than its details, except to the extent that the details must be gotten right in order to fit the pieces together. A useful methodology is more than a collection of separately good ideas. Thus the issue of combination re-occurs on the meta-level. There is not much that I can offer in the way of solid advice, other than the warning that it is harder than it looks to get all the details right.

It is too soon to draw any 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. While writing them, we uncovered several design errors by inspection; we are encouraged that many of these errors would have been uncovered by the checks called for in the language definition. However, until we have the tools that will allow us to gain experience with automated semantic checking, it is impossible to know just how helpful these checks will be.

We have not yet written any large specifications in Larch interface languages. Small examples seem to work out well. The Larch style of two-tiered specification leads to specifications that looks like they will "scale" well. We are presently in the process of documenting Larch/CLU, and are using it to write more substantial interface specifications. That experience should give us a much firmer basis for evaluating the Larch Shared Language, Larch/CLU, and—most importantly—the Larch style of specification.

Acknowledgements

The work that I have been describing was done in collaboration with John Guttag and his students at MIT—especially Randy Forgaard, Ron Kownacki, Jeannette Wing, and Joe Zachary. John's influence has been all-pervasive.

My ideas about formal specification have been shaped over the years by so many people that I hesitate to give an incomplete list. However, I am especially indebted to IFIP Working Group 2.3 (Programming Methodology), both for a continuing education and for being a constructively critical sounding board. I vividly recall getting key ideas during discussions with Jean-Raymond Abriel, Rod Burstall, Cliff Jones, Bill McKeeman, Doug Ross, Mary Shaw, Jim Thatcher, and Steve Zilles.

Jeannette Wing, Butler Lampson, and Soren Prehn have been especially helpful in improving the exposition.

The Larch Project has been supported at the Massachusetts Institute of Technology's Laboratory for Computer Science by DARPA under contract N00014-75-C-0661, and by the National Science Foundation under Grant MCS-811984 6, by the Digital Equipment Corporation at its Systems Research Center, and by the Xerox Corporation at its Palo Alto Research Center.

References

[ADJ 78] J. A. Goguen, J. W. Thatcher, and E. G. Wagner, "Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," in R. T. Yeh (ed.), *Current Trends in Programming Methodology, Vol. IV, Data Structuring*, Prentice-Hall, Englewood Cliffs, 1978.

[Burstall and Goguen 77] R. M. Burstall and J. A. Goguen, "Putting Theories Together to Make Specifications," *Proc. 5th International Joint Conference on Artificial Intelligence*, Cambridge, MA, 1977, 1045–1058.

[Burstall and Goguen 81] — , "An Informal Introduction to Specifications Using CLEAR," in R. Boyer and J. Moore (eds.), *The Correctness Problem in Computer Science*, Academic Press, New York, 1981, 185–213.

[Forgaard 84] R. Forgaard, "A Program for Generating and Analyzing Term Rewriting Systems," S.M. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, MIT/LCS/TR-99, 1984.

[Guttag 75] J. V. Guttag, "The Specification and Application to Programming of Abstract Data Types," Ph.D. Thesis, Computer Science Department, University of Toronto, 1975.

[Guttag and Horning 80] — and J. J. Horning, "Formal Specification as a Design Tool," *Proc. ACM Symposium on Principles of Programming Languages*, Las Vegas, Jan. 1980, 251–261.

[Guttag and Horning 83] — , "Preliminary Report on the Larch Shared Language," Technical Report MIT/LCS/TR-307 and Xerox PARC CSL-83-6, 1983.

[Guttag and Horning 85a] — , "Report on the Larch Shared Language," *Science of Computer Programming*, to appear.

[Guttag and Horning 85b] — , "A Larch Shared Language Handbook," *Science of Computer Programming*, to appear.

[Guttag and Horning 85c] — , "An Overview of the Larch Family of Specification Languages," in draft.

[Guttag, Horning, and Wing 82] — , and J. M. Wing, "Some Notes on Putting Formal Specifications to Productive Use," *Science of Computer Programming*, vol. 2, Dec. 1982, 53–68.

[Guttag, Horning, and Wing 85] — , "Preliminary Report on the Larch/CLU Interface Language," in draft.

[Hehner 84] E. C. R. Hehner, "Predicative Programming, Parts I and II," *Comm. ACM*, vol. 27, Feb. 1984, 134–151.

[Lescanne 83] P. Lescanne, "Computer Experiments with the REVE Term Rewriting System Generator," *Proc. ACM Symposium on Principles of Programming Languages*, Austin, Jan. 1983, 99–108.

[Musser 80] D. R. Musser, "Abstract Data Type Specification in the Affirm System," *IEEE Transactions on Software Engineering*, vol. 1, 1980, 24–32.

[Nyborg 84] *Proc. Workshop on Combining Specification Methods*, Nyborg, May 1984, Springer-Verlag.

[Wand 79] M. Wand, "Final Algebra Semantics and Data Type Extensions," *Journal of Computer and System Sciences*, vol. 19, 1979, 27–44.

[Wing 83] J. M. Wing, "A Two-Tiered Approach to Specifying Programs," Ph.D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, MIT/LCS/TR-299, May 1983.