Developing Formally Verified Ada Programs*

Norman Ramsey†

Odyssey Research Associates

October 10, 1988

Abstract

Odyssey Research Associates has undertaken a study of the feasibility of developing formally verified Ada programs. We have designed a specification language for sequential Ada programs. It is a member of the Larch family of specification languages. We have built a prototype program editor that is intended to help programmers develop programs and proofs from specifications, as advocated by Dijkstra and Gries [1, 2]. It contains predicate transformers, which compute *wp* (an approximation to the weakest precondition of a program), and it generates verification conditions.

The semantics of the specification language and the definition of the predicate transformers are derivable from a denotational definition of sequential Ada. The predicate transformers can be proved sound with respect to these definitions by structural induction on programs. The denotational-style definition of the predicate transformers is well suited to an implementation as an attribute grammar.

The program editor is designed to be used on program fragments, not just complete programs. The next step in improving the prototype editor is to find ways to simplify the intermediate values of *wp* so they can be used to guide the development of fragments into programs.

Introduction

Writing formal specifications of programs and proving that programs meet those specifications should help programmers develop more reliable software. Edsger Dijkstra and David Gries have stressed that a program should not face verification as a hurdle after development, but should be developed in such a way as to ensure its correctness [1, 2]. One should begin with a formal specification, and the development of the program and its proof should follow from that specification. Gries, in his *Science of Programming*, suggests ways to use a specification to guide the development of a program and its proof.

We have undertaken to build software that will help programmers apply the methods of formal development advocated by Gries and Dijkstra. The software is designed to lead to formal verification tools with three properties:

- The tools should not just help to verify finished programs or to check proofs of

*This research has been sponsored by the USAF, Rome Air Development Center, under contract number F30602–86–C–0071.

†Current address: Department of Computer Science, Princeton University, Princeton, New Jersey 08544

such programs, but should help programmers to develop verified programs.

- The tools should be based on sound, explicitly stated mathematics.
- The tools should support programming in a subset of Ada, since there is a need for reliable software written in Ada.

Overview of results

We have designed a specification language, Larch/Ada-88, for sequential Ada programs. We have implemented a prototype of an editor, Penelope, which will help programmers develop and verify programs specified with Larch/Ada-88. The prototype implementation supports a subset of Ada that is roughly "PASCAL with exceptions." We have completed some of the mathematics that supports the Larch/Ada-88 definition and the Penelope implementation.

Specification language The Larch/Ada-88 specification language is part of the Larch family of two-tiered specification languages [3, 4].[1] The two-tiered approach separates the specification of individual program modules from the specification of underlying abstractions. The Larch Shared Language is used to specify the underlying abstractions; for example, it can be used to define the notions of array, list, set, bag, and so on. A Larch Shared Language specification defines a set of terms (and some theorems about the terms); this set becomes the *assertion language* used in Larch/Ada-88 specifications.

Larch/Ada-88 (henceforth Larch/Ada) is a Larch *interface language*; it is used to specify Ada programs by attaching assertions at certain points, like the entry and exit points of subprograms. The specification constructs of Larch/Ada are called *annotations*, since most of them are derived from similar constructs in Anna [5].

Implementation Most verification sofware works in batch mode [6, 7]. The user writes a program, supplies a specification and appropriate assertions, and then submits it all to a *verification condition generator*. The resulting verification conditions must be shown to hold. A verification condition for a program is an assertion whose truth guarantees that the program satisfies its specification.

When a programmer using a batch system makes a refinement or discovers a mistake, the whole job must be resubmitted, and correct work may have to be redone. Moreover, most batch systems cannot verify program fragments. As practiced by Gries and Dijkstra, program development consists largely in building up correct fragments by accretion, and in refining existing fragments. Batch verification fits poorly with these techniques.

We have replaced the traditional batch verification system with a program editor, which we call Penelope. Using Penelope, programmers can examine weakest preconditions as they are computed and can use them to guide program development. Program fragments can be proved correct, then picked up and placed in larger contexts.

We have built the Penelope editor by using the Cornell Synthesizer Generator [8, 9]. Penelope can be used to create and edit abstract syntax trees that represent parts of an annotated Ada program. The interface is that of a traditional syntax-directed editor in which the user may use a mouse or an EMACS-like command set to manipulate trees. Penelope computes weakest preconditions as attributes

[1] We use "specification" in the traditional sense of a statement of requirements. What the Ada Language Reference calls "specifications" should be thought of as like "declarations" in other programming languages.

of the nodes of the syntax tree.[2] Attribute evaluation is incremental; that is, every time the user changes the tree, weakest preconditions are recomputed where necessary, and the new preconditions are available for display. When certain Ada constructs are used, Penelope generates a verification condition; typically one verification condition is generated for each subprogram and one for each loop.

A user of Penelope begins by writing down a formal specification of an Ada subprogram using Larch/Ada. (The formal specification takes the form of an Ada "specification" augmented with *subprogram annotations*.) He or she then builds the subprogram body, working backwards as described by Gries [2]. Penelope can be instructed to display weakest preconditions at any point, and also to display any verification conditions that may be generated. The user can alter the program and immediately observe the effects on preconditions and verification conditions, since the recomputation of *wp* and of verification conditions is incremental and automatic. (The preconditions and verification conditions are simplified somewhat before being presented to the user.)

The verification conditions are sentences in pure logic; their statement is independent of program context. If they can be shown to hold, the program containing them satisfies its specification. Penelope contains a proving component that can be used to prove facts about integers, Booleans, and Ada types like arrays and records. The proving component is primitive; the proof of a program like binary search takes up many pages.

Formal foundations The formal foundation of Larch/Ada has two parts, one dealing with assertions and another dealing with annotations. The assertion part covers the semantics of the Larch/Ada assertion language as defined using Larch Shared Language specifications. This semantics is determined entirely by the semantics of the Larch Shared Language, and is therefore independent of Ada. The assertion part also covers the formal specification (using the Larch Shared Language) of Ada's data types.

The annotation part connects the execution semantics of sequential Ada with the annotations of Ada programs defined by Larch/Ada. The Penelope editor's weakest precondition computations are based on a formal statement of *predicate transformers*. The predicate transformers define a function from a Larch/Ada specification and an annotated Ada program to a set of verification conditions. The transformers are based on a continuation semantics for the sequential part of Ada. As part of this work, Polak [10] has shown how to establish a formal connection between a continuation semantics and predicate transformers. To define the predicate transformers, he lets the denotation of a program be a function on terms of the Larch/Ada assertion language.

Larch/Ada

Two-tiered specifications In the two-tiered system of specification, the *shared language* component is used to define all the abstractions used in specification and verification. For example, in an arbitrary-precision arithmetic package, the shared language part of the specification would define what we mean by integer and by addition. One might implement arbitrary-precision arithmetic using the idea of registers of arbitrary length. The shared language would be used to define what we mean by a register and its length.

The *interface language* part of a specification is used to state what a program does in

[2] Actually, as in Gries [2], the system works with an *approximation* to the weakest precondition, called *wp*.

terms of the abstractions defined in the shared language part. The interface language part of the arbitrary-precision arithmetic package would show which subprograms perform what operations on the data, and would state that overflow never occurs.

Using the two-tiered system, a designer can keep the unpleasant details introduced by the programming language isolated in the interface language component, while doing the real intellectual work of specification in the shared language component. It is possible to use the shared language to say exactly what is meant by, for example, a stack or a directed graph, without getting bogged down in details of exceptional conditions or of representation.

The shared language part of a specification defines the assertion language used in the interface language part of the specification. Formulas in this assertion language are formulas in first-order predicate logic. When we attempt to prove that an implementation satisfies its specification, we first apply the predicate transformers, which are based on the definition of Ada, to a program and its specification. The predicate transformers produce a verification condition, which is a sentence in the assertion language. The proof of the verification condition needs to refer only to the shared language part of the specification; no further reference to the definition of Ada is required. Since the assertion language is really a particular formulation of first-order logic, checking the correctness of the proofs of verification conditions is straightforward.

Languages in the two-tiered system Three languages are important in the two-tiered system of specification. The Larch Shared Language enables users to write formal specifications of useful abstractions. It is a single language, used to write specifications. The Larch Shared Language part of a specification consists of one or more *traits*, each one of which may specify several abstractions. The shared language part of a specification defines an *assertion language*, which is the language used to refer to the abstractions. (A sentence stating that a stack is not empty would be a Boolean term in the assertion language.) In general, every shared language specification defines a different assertion language, although the different assertion languages have much in common, since they are just different formulations of first-order logic.

Larch/Ada is the language in which we write the *interface language* part of a specification. This part uses *annotations* to specify the behavior of a program. The annotations contain assertions, which are formulas in the assertion language defined by the shared language part of the same specification.

Description of Larch/Ada The assertion language defined by a particular shared-language specification is essentially a particular predicate calculus. We will describe the features of the Larch/Ada interface language, then give an example.

The simplest Larch/Ada annotation is the *embedded assertion.* The assertion is a formula in the current assertion language (defined by the shared language part of the current specification). Free variables in the assertion refer to program variables. Embedded assertions may be inserted at certain control points in an implementation (i.e. between statements or between declarations); they constrain the implementation to satisfy the assertion whenever control reaches that point. In other words, the embedded assertion says that, whenever control reaches it, if we substitute the actual values of program variables into the assertion, it must denote truth. The embedded assertion can be used only in an implementation, as a guide to the predicate transformers; it cannot be used in specifications.

Larch/Ada's *subprogram annotations* are used both to constrain subprogram implementations, and to specify subprograms. The entry/exit kinds are the *in annotation*, the *out annotation*, and the *result annotation*. Each of these contains a single assertion. They are used to constrain states on entry to a subprogram, to constrain states on exit from a subprogram, and to constrain the values a function might return.

Also among the subprogram annotations are several *exception propagation annotations*, which are used to say when exceptions may and may not be raised, when exceptions must be raised, and what conditions must hold when exceptions are raised. Finally there is the *side effect annotation*, which is used to document side effects on or dependence on global variables.

Larch/Ada has two features which help make the annotation mechanism more powerful. The first is the "`IN` variable." Within a subprogram, parameter names can be modified by `IN` to denote the value of the parameter on entry to the subprogram. `IN` variables are essential in subprogram specifications, because we almost always need to refer to the initial values of parameters. For example, here are specifications for some of the familiar stack operations:

```
PROCEDURE push(s: IN OUT stack; x: element);
--| WHERE
--|     OUT s = push(IN s,x);
--| END WHERE;

PROCEDURE pop(s: IN OUT stack);
--| WHERE
--|     IN NOT is_empty(s);
--|     OUT s = pop(IN s);
--| END WHERE;
```

The other enriching feature of Larch/Ada is a way to introduce "virtual variables."[3] These variables are not used in writing specifications, but are defined within implementations to help in proofs of correctness. Their values don't actually affect the results of a computation. Larch/Ada allows the user to declare virtual variables, to assign to them, and to use them in annotations.

Finally, in support of data abstraction (packages with private types), Larch/Ada enables the user to define *abstraction functions* using the Larch Shared Language, and to associate these abstraction functions with Ada types, using the `based on` annotation. Thus, a user writing an arbitrary-precision arithmetic package might implement registers using arrays, and would then define an abstraction function from arrays to registers. The proofs of correctness of implementations of abstract data types are as described by Hoare [11]; the abstraction function is used to rewrite a specification that was in terms of an abstract type to a new specification in terms of a concrete type.

Using Larch/Ada As an example, here is a Larch/Ada specification for some set operations. Jeannette Wing used these operators in her presentation of a Larch interface language for CLU [4]. As she did, we will use a Larch Shared Language description of sets and set operations. The trait describing these operations, called `SetOfE`, is shown in Figure 1.

[3]These are called "ghost variables" by Gries and Dijkstra; the name "virtual variable" is from Luckham [5].

We begin our example with the specification of a "choose" procedure that selects a member of a set, removes it from the set, and returns it. Since Ada functions may not have side effects on their parameters, we will formulate the returned member as an `OUT` parameter:

```
--| WITH SetOfE WITH [set FOR si, integer FOR e];
PROCEDURE choose (IN OUT s: set; i: OUT integer);
--| WHERE
--|     IN NOT IsEmpty(s);
--|     OUT has(IN s, i) AND s = remove(IN s, i);
--| END WHERE;
```

The `--| WITH` annotation specifies that `SetOfE` is the trait that defines the notion of set and the operations `IsEmpty`, `has`, and `remove`. The `IN` annotation states that `choose` may be called only on nonempty sets. The `OUT` annotation gives the relation between the initial and final values of the set `s` and the final value of the integer `i`. (Notice that the final values are specified by giving just the variable name, while the initial values are specified by modifying the name by `IN`. In the `IN` annotation, all variables are implicitly modified by `IN`.) Because no side effect annotation is present, `choose` may not modify or read any global variables.

Figure 2 shows a Larch/Ada specification for a set package. This package specifies the same operators as the similar example in Wing [4].

Formal foundation of Larch/Ada and Penelope

Connecting Ada to a denotational model Formal verification of Ada programs must be based on a formal definition of the Ada language itself, but at this time there is no official formal definition of Ada. We circumvent this difficulty by providing a denotational model of a computing language Ada′, and by considering Larch/Ada to be a specification language for Ada′. Ada and Ada′ have the same syntax, and we argue informally that for a restricted class of programs and computations they have the same observable behavior.

We restrict Ada most by omitting from Ada′ all features involving concurrency. While there is widespread consensus on what are good methods to model and specify sequential imperative languages, there is no similar consensus on the utility of the various proposed methods of modeling and specifying concurrent programs. We have omitted other features from Ada′ because they are machine-dependent (e.g. representation clauses) or because it is not feasible to formalize them (e.g. the exact circumstances under which `storage_error` is raised). Here is a partial list of omitted features:

- Concurrency
- Real number types
- Representation specifications and other features that are machine-dependent or implementation-dependent
- Unchecked conversion and unchecked deallocation
- The predefined exceptions `storage_error` and `numeric_error` (i.e. we consider that no execution of an Ada′ program ever raises these exceptions), and computations that result in "undetected numeric overflow"
- Optimizations that cause execution of Ada statements in other than the canonical order
- Parameter aliasing in procedure calls
- Any program called *erroneous* by the Ada reference manual

Some of these restrictions (e.g. that forbidding aliasing) can be enforced by suitable static checks.

Although Ada$'$ is not Ada, they are intended to be equivalent within our area of interest, and we will not distinguish them in what follows.

Connecting the denotational model to Larch/Ada and the predicate transformers The predicate transformers implemented in Penelope are derived from a continuation semantics for Ada. The task of defining a continuation semantics for Ada has been considerably simplified by two expedients. First, the static semantics of Ada is not part of the definition; the definition assumes a suitably checked and attributed abstract syntax representation of programs. Second, the semantics of the Ada types is not part of the definition; instead, the semantics of Ada types is defined by Larch Shared Language specifications. The technique used for deriving predicate transformers from a denotational semantics is one developed by Polak [12, 10]. The current definition of the predicate transformers used in Penelope does not provide for proofs of termination.

Here we give an example that shows what we mean when we say that Larch/Ada and Penelope are formally based. The example suggests how we define the semantics of Larch/Ada and how we show that the VC generation implemented in Penelope is sound.

Generating verification conditions involves manipulating a number of different languages. Since a VC generator takes as input a program and a specification, and produces as output an assertion (the verification condition), the fundamental languages are the programming language P, the specification language S, and the assertion language A. For simplicity, we take the assertion language as fixed, although it is actually determined using the shared language part of the specification.

The actual input to Penelope is a specification together with an *annotated* implementation; we will call that conglomerate V, and we define projection functions that extract the relevant parts:

$\pi : V \rightarrow P$	The implementation *without* annotations
$\sigma : V \rightarrow S$	The specification

Then the VC generator is a function $vcgen : V \rightarrow A$, and we wish to show that $vcgen$ is sound, i.e. whenever the verification condition $vcgen(v)$ holds, the program πv satisfies its specification σv.

We need to consider the denotations of the various syntactic objects we have been discussing. For simplicity, we'll let the denotation of a program be a mapping from states to states (this makes sense even for a continuation semantics if one considers whole programs). We'll call B the special Boolean domain consisting of the two elements truth and falsehood (which we'll write as $\mathbf{t}$ and $\mathbf{f}$). If the set of states is X, we have

$M_P : P \rightarrow (X \rightarrow X)$
A program denotes a state changer.

$M_S : S \rightarrow ((X \rightarrow X) \rightarrow B)$
A specification denotes a predicate on state changers.

$M_A : A \rightarrow (X \rightarrow B)$
An assertion denotes a predicate on states.

Intuitively, a program is a function from states to states, a specification defines a predicate ("satisfaction") on programs, and an assertion defines a predicate on states (also called "satisfaction"). A program $p \in P$ satisfies a specification $s \in S$ if

$$(M_S s)(M_P p) = \mathbf{t}.$$

The semantics of the specification language can be defined in terms of the semantics of the programming and assertion languages, M_P and M_A, provided we have a fixed (not necessarily finite) set of states X. If the specification language is a simple one that gives only entry and exit assertions,
i.e. $S = A \times A$, then we can define

$$M_S\langle a_1, a_2\rangle\rho = \forall x \in X. M_a a_1 x \Rightarrow M_a a_2(\rho x),$$

where $\rho = M_P(p)$ is a state transformer. The definition says that a program satisfies its specification if, whenever the entry condition a_1 holds on entry to the program, the exit condition a_2 holds on exit from the program. (We are using $\Rightarrow$ to denote mathematical implication.)

The semantics of Larch/Ada is defined in two steps: first, we define a mapping from Larch/Ada to a simpler language in which a specification consists of an entry condition, an exit condition for normal termination, and exit conditions for termination by raising exceptions. In the second step we define the semantics of the simpler language; this language is essentially $S = A \times \textbf{list}\ A$, and the definition of its semantics is very similar to that just shown.

As the discussion above suggests, V is not really fundamental; we introduced V to stand for the input to $vcgen$. If we have an (annotated program, specification) pair $v \in V$, we are really only interested in the projections πv and σv. The program satisfies its specification when

$$M_S(\sigma v)(M_P(\pi v)) = \textbf{true}.$$

VC generation is sound if the truth of the verification condition guarantees that the program satisfies its specification. The truth of a verification condition must be independent of state, so the function $vcgen$ is sound if, for any $v \in V$,

$$\forall x \in X(M_A(vcgen(v))) \Rightarrow M_S(\sigma v)(M_P(\pi v)).$$

Polak [10] goes into more detail, giving a complete definition of $vcgen$ for a small programming language P. He describes a way of deriving $vcgen$ from the semantics of P and sketches a proof of soundness that uses structural induction on programs. The techniques he describes are the same ones used to define the semantics of Larch/Ada and to prove the soundness of the predicate transformers implemented in Penelope.

The Penelope implementation

We are implementing tools that support research in formal verification using Larch/Ada. Of these, the most important is the Penelope editor, which helps users develop verified programs. In the prototype, there is no support for writing traits; every specification has the same shared language part, and there is a fixed assertion language used for all specifications. The prototype can be used to specify and prove Ada subprograms, provided those subprograms use no global variables. We plan to extend the Penelope prototype to read definitions of extensions to the assertion language. We plan to supplement it with a tool that will help prove Penelope verification conditions.

Status of the implementation

Penelope is implemented using the Cornell synthesizer generator [8]. The Cornell synthesizer generator accepts as input a description of an attribute grammar and compiles this description into a syntax-directed editor which can compute and display the values of attributes. The heart of this editor is an algorithm which, when the edited tree is changed, computes and propagates the changes in attribute values [9].

We think of Penelope as having three components: predicate transformation, proving, and simplification. Predicate transformation is central. That component reads and interprets the Larch/Ada annotations, computes wp, and generates verification conditions. The user controls which intermediate values of wp are displayed and which verification conditions are displayed. The displayed values are updated every time the user changes his or her program or specification.

The proving component is a sub-editor that enables the user to construct proofs of the verification conditions, using a sequent calculus. The editor presents a list of hypotheses and a goal, and the user designates an inference rule to apply. The application may generate subgoals, and the process continues until the subgoals are reduced to axioms, which are automatically recognized by the editor. The editor has built in a small number of proof tactics; the user can designate one of these tactics instead of designating a rule.

The simplification component is a set of functions that can be called by the other two components. These functions make the preconditions and verification conditions more readable. One function does this by rewriting; terms like $P \wedge \mathbf{true}$ are rewritten to P, and so on. Another function attempts to find ways to substitute simpler terms for terms that are especially complex or hard to read. Another performs arithmetic operations on integer literals. Several functions manipulate the forms of terms in order to make other operations easy; depending on circumstances, one may prefer $P \supset (Q \supset (R \supset S))$ to $P \wedge Q \wedge R \supset S$, or vice versa. (We are using $\supset$ to represent the implication symbol in the assertion language.) Collectively, these functions reduce the size of the verification conditions Penelope generates by about a factor of five.

The currently supported subset of Ada

We have imposed a number of restrictions on programs editable with Penelope, as described earlier. Concurrency, real number types, parameter aliasing, `storage_error`, and some other features are forbidden. Here are the highlights of what Penelope does with those programs it accepts:

- Subprograms may call predefined or user-defined subprograms; recursion is supported.
- Arbitrary user-defined exceptions, `raise` statements, and exception handlers are supported.
- Integer, Boolean, enumeration, array, and record types are supported. The predefined Ada operators for these types are supported. (Subtyping is not supported.)
- All of the Ada control structures are supported except `goto` statements, `case` statements, and `for` loops.
- Some static semantic checking is performed, including type checking and overload resolution.

The current Penelope is limited in what it can prove. Many capabilities which might be considered difficult have been investigated mathematically but have not yet been implemented. These include:

- Proving that a program never raises the predefined exceptions `constraint_error` and `program_error`

- Proving that programs terminate
- Proving that programs are not erroneous
- Detecting potential aliasing and illegal order dependencies by suitable static semantic checking
- Proving programs that define subtypes whose bounds are set dynamically
- Proving programs that raise or handle the predefined exceptions `constraint_error` and `program_error`

The currently supported subset of Larch/Ada The current implementation of the Penelope editor supports most of Larch/Ada. The major features that are missing are those associated with termination, global variables and side effects, and data abstraction. As noted earlier, there is no support for writing traits in the Larch Shared Language. This means that the shared language part of every specification is the same, and that there is a single assertion language used for all Larch/Ada specifications. That assertion language is restricted to terms describing integers, Booleans, arrays, and records. (Enumeration literals are converted to integers during predicate transformation.)

Future plans

Simplification The greatest weakness of the current Penelope editor is that weakest preconditions and verification conditions are too hard to read. The simplification component is good at reducing the bulk and complexity of Boolean terms; most of the complexity in verification conditions comes from arithmetic terms. Simplifying such terms is the next step in improving our implementation. Rather than build an arithmetic simplifier from scratch, we plan to connect an existing simplifier to the Penelope editor. The simplifier that we plan to use is based on the Nelson-Oppen procedure for combining decision procedures [13, 14].

Extending Penelope In order to support data abstraction, we intend to add packages and private types to Penelope's Ada subset. (Adding these constructs is straightforward; the major difficulties involved in supporting data abstraction arise in extending the assertion language, as discussed below.) We may also add new control structures, in particular the `case` statement and the `for` loop.

In the longer term, we plan to add support for proofs of termination and for proofs of subprograms that have side effects on global variables.

Data Abstraction It is not possible to write readable formal specifications of large programs without taking advantage of data abstraction. Generating verification conditions for programs that use abstract data types is not hard, but generating sound verification conditions for implementations of abstract data types can be tricky. This is especially true in Ada, where the abstraction constructs do not completely hide the representation.

It is also hard to simplify and prove verification conditions when specifications refer to abstract data types. The essence of the difficulty is that, when using data abstraction, the user must add new terms to the assertion language. (These terms describe the new abstractions, like stacks, buffers, registers, or whatever may be needed to specify a particular application. They are introduced and defined by traits written in the Larch Shared Language.) To be able to make effective use of the new terms, we must be able to show that their introduction does not lead to any logical inconsistency. We must also be able to extend the proving and simplification components of Penelope to be able to handle the new terms.

We will begin studying data abstraction by making our assertion language extensible. We will use a tiny subset of the Larch Shared Language, a subset which will enable us to add to the assertion language new sort and operator symbols. In particular, it will be possible to add abstract sorts and abstraction functions to the assertion language. We will then allow users to make assertions (without proof) involving the new symbols they have introduced. In proofs of programs, these assertions will be treated like axioms. We hope that, by studying the kinds of assertions users make, we will be able to learn what methods of proof might help users prove programs that use data abstraction.

Conclusions

Our efforts have been concentrated on defining Larch/Ada-88 and on building the Penelope prototype. Evaluation of Larch/Ada and Penelope must await the completion of the prototype and experience with its use, but we can draw some conclusions about the methods we have applied and about the difficulty of the problems that remain.

We have developed a useful technique for deriving predicate transformers, and we have developed a method for implementing the transformers using an attribute grammar. We have some preliminary observations about the results of attempting to mechanize Gries's and Dijkstra's methods of program development. Finally, we believe we have learned what problems need to be solved before a useful verification system can be built.

Implementing predicate transformers The denotational style of writing predicate transformers lends itself to a natural and efficient implementation of the transformers as an attribute grammar. Values in the transformer definition map to attributes of the grammar, and meaning functions map to the semantic equations that define the relationships among the attributes. We can avoid implementing lambda-abstraction and beta-reduction for the language of terms by using pairs of attributes to represent values of arrow types.

Mechanizing formal development The biggest obstacle to learning Gries's method of program development is the drudgery of computing *wp*. This difficulty increases as the complexity of the programming language increases; it would be unrealistic to expect to compute *wp* by hand for a language like Ada. Fortunately, it is easy to mechanize the computation of *wp*, provided a denotational-style definition of *wp* is available.

The problem with a mechanized computation of *wp* is that the resulting preconditions quickly become too complicated to be understood by a human being, at which point they can no longer be used to guide program development, which is the whole point of Gries's method. We have yet to learn whether mechanical simplifiers like the one described by Nelson and Oppen can make the preconditions understandably simple. The problem doesn't arise when *wp* is computed by hand, because in that case the programmer constantly applies his or her knowledge of integers, sequences, Booleans, and so forth, so that computation and simplification proceed simultaneously.

Open problems The Larch/Ada specifications we can write using Penelope are limited by the fixed, nonextensible assertion language. (It is a severe limit; for example, at this time we cannot introduce the factorial function for use in a specification.) The programs we can prove using Penelope are limited by the size of the weakest preconditions Penelope computes. We believe that the most important problem remaining to be solved is the one of being able to introduce new terms into an assertion language, while simultaneously introducing methods of simplification and proof for those terms. The Larch Shared Language provides a way of writing formal definitions of new terms. We need to develop a formal representation of methods of proof and simplification. Finally, we need to develop ways of showing that the addition of new definitions introduces no logical inconsistency, and ways of showing that the proof and simplification methods are consistent with the definitions.

Related work

Researchers at MIT and DEC SRC have developed several tools that manipulate Larch Shared Language traits. The Larch prover, LP, can read a description of an equational theory and attempt to prove sentences in that theory, primarily by term rewriting [15]. The Larch checker, LC, reads a trait and writes two files for input to LP: an axiomatization of the trait and a command script for discharging the proof obligations associated with that trait. (Not all of the proof obligations described in the reference manual for the shared language have been implemented yet.) LC supports the shared language's mixfix and overloaded operators [16].

AFFIRM, built at USC-ISI, was the first verification system to use both algebraic specifications and a rewrite rule prover [17]. The Gypsy system was the first verification system to handle a form of concurrency [6]. The Stanford Pascal Verifier was the first verification system to handle a real programming language [18]. The most important contribution of the Stanford Pascal Verifier project was probably the Nelson-Oppen method of combining decision procedures [13].

The Anna project is an effort to introduce formal specification to Ada programmers by providing specification constructs which can be checked at run time [5]. The aim of AVA project is to define a verifiable subset of Ada and to give it a formal semantics using Boyer-Moore logic [19, 20]. The AVA language is essentially a subset of that implemented in the Penelope prototype; it omits exception handlers and `raise` statements.

Acknowledgements

The work described herein was done at Odyssey Research Associates with Wolfgang Polak, Carla Marceau, David Guaspari, C. Douglas Harper, and Doug Weber. Anna provided large repository of specification constructs, on which we drew heavily when designing Larch/Ada. The Anna group at Stanford was forthcoming with suggestions about how formal verification of Ada programs might proceed. John Guttag helped us to understand Larch and to explore how we might adapt Larch to Ada. In particular, he helped us elucidate the issues that needed to be addressed in defining the

semantics of a Larch interface language. John Guttag and Steve Garland lent us LP, which we used in our study of verification conditions generated by Penelope.
David Guaspari and Wolfgang Polak helped me understand the formal foundations of Larch/Ada and Penelope, and they corrected many errors in earlier drafts of this paper. Of course, they are not responsible for any errors that may remain.

References

[1] E. W. Dijsktra, *A Discipline of Programming.* Prentice-Hall, 1976.

[2] D. Gries, *The Science of Programming.* Springer-Verlag, 1981.

[3] J. V. Guttag, J. J. Horning, and J. M. Wing, "Larch in five easy pieces," Tech. Rep. TR 5, DEC/SRC, July 1985.

[4] J. M. Wing, "Writing Larch interface language specifications," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 1–24, Jan. 1987.

[5] D. C. Luckham *et al.*, "Anna: A language for annotating Ada programs," tech. rep., Stanford University, 1986. The Anna reference manual.

[6] D. I. Good, R. L. Akers, and L. M. Smith, "Report on Gypsy 2.05," tech. rep., Computational Logic Inc., 1986.

[7] J. Crow, S. Jefferson, R. Lee, M. Melliar-Smith, J. Rushby, R. Schwartz, R. Shostak, and F. von Henke, "Preliminary definition of the revised SPECIAL specification language," tech. rep., SRI International, 1986.

[8] T. Reps and T. Teitelbaum, *The Synthesizer Generator Reference Manual.* Department of Computer Science, Cornell University, 1987.

[9] T. Reps, *Generating Language-Based Environments.* MIT Press, 1984.

[10] W. Polak, "A technique for writing predicate transformers." Unpublished manuscript.

[11] C. A. R. Hoare, "Proof of correctness of data representations," *Acta Informatica*, vol. 1, pp. 271–281, 1972.

[12] W. Polak, "Program verification based on denotational semantics," in *Principles of Programming Languages*, 1981.

[13] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *ACM Transactions on Programming Languages and Systems*, vol. 1, pp. 245–257, Oct. 1979.

[14] T. Redmond, "Simplifier description," Tech. Rep. ATR-85 (8354)-8, Aerospace, Nov. 1985.

[15] S. Garland and J. V. Guttag, "An overview of LP, the Larch prover," in *Third International Conference on Rewriting Techniques and Applications*, 1988.

[16] S. Garland. Private communication.

[17] D. R. Musser, "Abstract data type specifications in the AFFIRM system," in *Proceedings of the Specifications of Reliable Software*, pp. 47–57, IEEE Computer Society, Apr. 1979.

[18] D. C. Luckham *et al.*, "Stanford Pascal Verifier user manual," Tech. Rep. STAN-CS-79-731, Stanford University, Mar. 1979.

[19] M. K. Smith, “A verifiable Ada,” *Ada Letters*, vol. 8, pp. 136–142, July/August 1988. In the Formal Methods Committee report.

[20] R. S. Boyer and J. S. Moore, “Proving theorems about LISP functions,” *Journal of the ACM*, vol. 22, no. 1, pp. 129–144, 1975.

Figure 1: Shared language specification of set operations (Wing [4])

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

Figure 2: Larch/Ada specification for a set package

```
--| WITH SetOfE WITH [integer FOR e];
PACKAGE sets IS
  TYPE set IS PRIVATE; --| based on si;

  FUNCTION pair(i, j: integer) RETURN set;
    --| WHERE
    --|   RETURN add(add(empty, i), j);
    --| END WHERE;

  PROCEDURE union(s1: set; s2: IN OUT set);
    --| WHERE
    --|   OUT (FORALL j::
    --|          ((has(s2, j) = has(IN s1, j))
    --|                       OR has(IN s2, j)));
    --| END WHERE;

  PROCEDURE intersect(s1: set; s2: IN OUT set);
    --| WHERE
    --|   OUT (FORALL j::
    --|          ((has(s2, j) = has(IN s1, j))
    --|                      AND has(IN s2, j)));
    --| END WHERE;

  FUNCTION member(s: set; i: integer) RETURN boolean;
    --| WHERE
    --|   RETURN has(s, i);
    --| END WHERE;

  FUNCTION size(s: set) RETURN integer;
    --| WHERE
    --|   RETURN card(s);
    --| END WHERE;
END sets;
```