A Methodology for Formal Specification and Implementation of Ada Packages Using Anna

Neel Madhav* and **Walter Mann**†

Program Analysis and Verification Group
Computer Systems Laboratory
Stanford University
Stanford, California 94305–4055

Computer Systems Laboratory Technical Report CSL–TR–90–438

Program Analysis and Verification Group Report No. 50

August, 1990

Abstract

This paper presents a methodology for *formal specification* and *prototype implementation* of Ada packages using the Anna specification language.

Specifications play an important role in the software development cycle. The methodology allows specifiers of Ada packages to follow a sequence of simple steps to formally specify packages.

Given the formal specification of a package resulting from the methodology for package specifications, the methodology allows implementors of packages to follow a few simple steps to implement the package. The implementation is meant to be a prototype.

This methodology for specification and implementation is applicable to most Ada packages. Limitations of this approach are pointed out at various points in the paper.

We present software tools which help the process of specification and implementation.

Keywords—*Ada, Ada Package, Anna, formal specification, prototype implementation, sufficient completeness*

*Department of Computer Science, Stanford University, Stanford, California 94305. E-mail: Madhav@Cs.Stanford.Edu

†E-mail: Mann@Anna.Stanford.Edu

1 Introduction

Formal specifications are a precise and mathematical description of what a program is supposed to do [1, 6].

The motivation for using specifications is quite clear [6]. Specifications allow a software manager, through a specifier, to put thoughts about what a program is supposed to do in a formal and precise language. The specification thus acts as a *blueprint* for the program. Specifications allow the *verification* [2, 7] or *testing* [4] of the correctness of a program (with respect to its specification). Specifications also help in implementing a program, as our methodology for implementations demonstrates. The process of automatically generating an implementation, given a specification, is called *automatic programming* [12].

This paper presents a methodology for *formal specification* and *prototype implementation* of Ada [15] packages using the Anna[9] specification language. Anna is a language for formally specifying Ada packages.

Section 2 introduces Anna formal specifications.

Section 3 presents a methodology for producing a formal specification of a module, given its interface. The interface of a module is made up of subprograms that the module provides to its clients. We start from an Ada package specification and obtain a formally specified Anna package. In the following we distinguish between Ada package specifications (the interface made up of subprograms) and Anna formal specifications (behavioral description).

The methodology can be applied to any module specification. It extends the work of [3] by allowing a more general framework of specifications, including abnormal situations. The methodology results in a specification which is provably *sufficiently-complete* [3]. The level of confidence in the consistency of the specification is also considerably increased.

The methodology may not result in the best specification. In certain cases the application of the methodology may be restricted due to the fact that the intended behavior of a program may not be apparent in the format required by the methodology. However, for a wide range of programs the methodology does result in a formal specification.

Section 4 presents a tool, the Anna Specification Analyzer [10], which allows the correctness of a specification to be tested.

Section 5 considers the question of obtaining a *prototype* implementation, given a formal specification of a package [13]. Given a formally specified package resulting from the methodology for specifications, the methodology for implementations allows an implementor to follow a few simple steps to get a prototype implementation.

Consider the following example:

Example 1:

$$\text{Square\_Root}(X) * \text{Square\_Root}(X) = X$$

This specification describes precisely what the function does but gives absolutely no clue as to how an implementation may be attempted. In general, the problem of getting even a prototype implementation, given a specification, is difficult. Our restricted framework of specifications does allow a prototype in most cases.

The resulting implementation is typically not the most efficient. However, the prototype implementation allows considerable insight to the properties of the final implementation and may act as the starting point for the final implementation.

Section 6 outlines how tools like verifiers and program testing tools [14] can be used to prove and/or test the correctness of the final implementation.

2 Specifying Ada Packages Using Anna

Packages are an Ada construct used to encapsulate logically related entities. Ada packages have two parts—the *specification* which defines the interface of the package and a *body* that defines the implementation. Packages can have types, objects, subprograms, and exceptions.

Anna is an extension of Ada to include formal specification of the intended behavior of Ada programs. There are constructs available in Anna for specifying each part of an Ada package. Anna constructs are embedded in Ada comments and are preceded by "--|".

The rest of this Section describes a few Anna formal specifications which can be used to specify normal behavior of subprograms, abnormal behavior of subprograms, and behavior of packages as a whole. Anna has other constructs which may be used to formally specify Ada constructs like subtypes and objects which are not discussed below. The discussion of Anna below is thus not meant to cover all aspects of Anna, just those that deal with formal specification of packages.

2.1 Subprogram Annotations

Subprogram annotations constrain the input and output behavior of subprograms. The keywords *in* and *out* distinguish the input and output values of parameters of procedures. Results of functions are constrained by *result* annotations.

Example 2:

```
procedure Minus(X, Y : in out Positive);
--| where (in X >= in Y),
--|          out (X = in X - in Y);
-- Input value of X should be greater than or equal to Y. The output
-- value of X should be the difference of input values of X and Y.
function Square(X : in Integer) return Integer;
--| where return Y : Integer => Y = X * X;
```

2.2 Exception Propagation Annotations

Strong exception propagation annotations describe conditions under which an exception will be raised. *Weak* exception propagation annotations describe conditions which must be true when an exception is propagated out of a subprogram.

Example 3:

```
procedure Quotient(X, Y : in out Float);
--| where (in Y = 0.0) => raise Divide_Error,
--|            raise Divide_Error => out (in X = X);
-- If the input value of Y is 0 then exception Divide_Error is raised. If
-- the exception is raised, the value of X remains unchanged.
```

2.3 Axioms

Axioms are first-order predicate logic constraints on the behavior of a package as a whole. Axioms describe relationships between subprograms of a package.

Example 4:

```
package Stacks is
      type Stack is private;
      function Push(S : Stack; E : Integer) return Stack;
      function Pop(S : Stack) return Stack;
      ...
      --| axiom for all S : Stack; E : Integer =>
      --|        Pop(Push(S,E)) = S;
end Stacks;
-- A sequence of a Push followed by a Pop should leave a stack un-
-- changed.
```

2.4 A Note on Definedness

An expression is said to be *defined* if its evaluation terminates normally. Abnormal behavior may be divided into two categories—exceptional termination and non-termination.

Subprogram annotations which constrain *in* values of parameters of subprograms allow the specification of abnormal behavior. For example, in Example 2 above, the subprogram Minus may display either kind of abnormal behavior if the value of X is less than Y.

Exception propagation annotations allow the specification of exceptional behavior of programs. Anna thus allows a specifier to distinguish between the two kinds of abnormal behavior.

3 A Methodology for Specifying Ada Packages

We now present a methodology for obtaining a formal Anna specification given an Ada package specification.

We start with a methodology for a subset of Ada packages and then show that packages outside the subset can be rewritten as packages in the subset. Ada package specifications in the subset we consider do not have any procedures or objects and do not have an internal state. The form of such a package specification then, is a series of type declarations followed by declarations of functions which act on these types.

Consider a package specification of lists. This is a generic package with a generic formal parameter, Item.

Example 5:

```
generic
     type Item is private;
package List_Package is
     type List is private;
     Out_Of_Range : exception;
     function Length(L : List) return Natural;
     function Get(L : List; N : Positive) return Item;
     function Create return List;
     function Insert(L : List; I : Item;
                     After : Natural) return List;
     function Delete(L : List; N : Positive) return List;
     function Member(I : Item; L : List) return Boolean;
private
     . . .
end List_Package;
```

The aim now is to provide Anna formal specifications which describe the result of executing each subprogram of the package. We extend the methodology presented in [3]. The set of functions is divided into *generators* which build values of new types defined in the package and *observers* which map values of new types to old types. The idea is to define the effect of each generator on each observer. This results in a specification which is *sufficiently complete* [3]. Sufficient completeness implies that the axioms have conveyed the full meaning of the operations.

Step I Identify the set of *new types* introduced (declared) in the package. Call this set $NewT$. For the list package this set has a single new type, List.

Step II Identify the set, Obs, of *observers* among the functions of the package which have a return type T not in $NewT$. For the list package this set has the functions Length, Get and Member.

For each type $T \in NewT$, identify the set $Obs(T)$, called the set of observers of type T, of functions in Obs at least one of whose parameters is of type T. For the list package $Obs(List) = Obs$.

Step III Identify the set, Gen, of *generators* among the functions of the package which have a return type $T \in NewT$. For the list package this set has the functions Create, Insert and Delete.

For each type $T \in NewT$, identify $Gen(T)$, called the set of generators of type T, of functions in Gen which have the return type T. For the list package, $Gen(List) = Gen$.

Step IV Choose the set, $ObsBas(T)$, of *basic observers* of each type $T \in NewT$. Basic observers are functions in $Obs(T)$ which cannot be expressed in terms of other observers. A useful condition to consider is that if $Exp1$ and $Exp2$ are expressions of type T then $f(Exp1) = f(Exp2)$ for all basic observers f, iff $Exp1 = Exp2$. In other words, two values of a type are equal iff the basic observers cannot distinguish between them.

For the list package this set has Length and Get. Get itself does not form an observer basis since it cannot distinguish between a list and any sublist with the first n ($n < Length$) elements of the list. This holds due to that fact that Get will either return the same element for both lists or return an exception for one list. Equality in universally quantified axioms is defined such that two expressions are equal if any of them does not terminate normally.

Member can be expressed in terms of the other two observers as shown below. Member and Length do not make a set of basic observers since they cannot distinguish between two lists of the same length which have a re-occurring item.

In general, the choice of basic observers is not unique. In practice, it is not difficult to identify a set of basic observers. It is not crucial for the following treatment to obtain a minimal set of basic observers. For example, Member could have been (erroneously) chosen a basic observer without substantially affecting the methodology.

Step V Choose the set, $GenBas(T)$ of *basic generators* of each type $T \in NewT$. Basic generators are functions in $Gen(T)$ which cannot be expressed in terms of other generators. The condition on basic generators is that *any* expression Exp of type T should be equal to an expression made up of just basic generators. In other words, basic generators generate all values of the type.

For the list package this set has Create and Insert. Delete is not a basic generator because any list that can be generated using Delete can be generated using just Create and Insert. Create and Delete do not make a set of basic generators since any list with non zero length cannot be generated just using Create and Delete.

Like basic observers, the set of basic generators is not unique and again it is not crucial to the methodology that a minimal set of basic generators be chosen.

Step VI For each type $T \in NewT$ and for each pair of functions $o \in ObsBas(T)$ and $g \in GenBas(T)$, define the effect each o has on each g. Each new expression $g(X)$ of type T built by application of a generator g may change what the observers return. Therefore *all* expressions which look like $o(g(X))$ should be set equal to some other known expression (which does not involve type T). The general form of such expressions is $o(g(X*, Y*), Z*)$ where $X*$ are variables of type T, $Y*$ are variables of other types, $Z*$ are variables which stand for argument types t_i of $o : t_1, \ldots, T, \ldots, t_n \rightarrow X$ and, $g(X*, Y*)$ is substituted for *one* of the parameters of type T. There are five options for how $o(g(X))$ can be defined:

1. $o(g(X))$ is equal to some expression $o(X)$. For example, in the list package :

```
--| Length(Create) = 0,
--| Length(Insert(L,I,N)) = 1 + Length(L),
--| Get(Insert(L,I,N),M) =
--|       if N + 1 = M then I
--|       elsif M <= N then Get(L,M)
--|       else Get(L,M-1) end if,
```

The expression $Get(Create, N)$ will be discussed later. A very simple inductive argument shows that the effect of basic generators on the observer basis is thus defined.

2. o(g(X)) is defined by a subprogram result annotation on g. The same approach is taken by [5, 11]. This is equivalent to the previous step but allows a more local specification. For the list example :

```
function Create return List;
--| where return L : List => Length(L) = 0;

function Insert(L : List; I : Item;
                After : Natural) return List;
--| where return R : List =>
--|       Length(R) = Length(L) + 1;
```

Get can be specified in the same way as Length.

3. $o(g(X))$ does not terminate normally. There are two options in this case—a subprogram annotation which restricts the domain that o can act upon, or a strong exception propagation annotation which specifies that an exception should be raised if input to o is $g(X)$.

```
function Get(L : List; N : Positive) return Item;
--| where in(Length(L) >= N);
-- Or.
--| where (Length(in L) < N) =>
--|       raise Out_Of_Range;
```

The function Get is permitted to not terminate or terminate with an exception in the first case. In the second case, the function must raise the named exception.

4. Define equivalence classes of generators. Take Integers for example :

```
package Integers is ...
    function Zero return Integer;
    function Succ(I : Integer) return Integer;
    function Pred(I : Integer) return Integer;
    ...
end Integers;
```

The functions Zero, Pred and Succ form the set of basic generators. However they do not uniquely generate any element, for example $X = Succ(Pred(X))$. This equation creates equivalence classes of Integer expressions. Observers need not be defined for each member of any equivalence class, it is enough that they be defined on any representative of the equivalence class.

5. Leave the expression o(g(X)) undefined. This results in an incomplete specification. However, the incompleteness is well defined in the sense that one can pinpoint the location of the incompleteness.

Step VII Define each observer $o \in Obs(T) - ObsBas(T)$ (which has not yet been defined) in terms of the defined observers *or* the defined generators. The specification may be easier to implement if the observers *or* the generators (but not both) are chosen to define all other functions in this step and in step **VIII**.

There are two options. Either, for each o, axioms of the form $o(X) = o2(X)$ should be given where $o2 \in$ defined observers. Or, $o(g(X))$ should be defined in terms of other expressions, for all $g \in$ basic generators. The method followed is the same as step **VI**. For example:

```
--| Member(I,L) = (exist N : 1 .. Length(L) =>
--|                         Get(L,N) = I),
-- Or.
--| not(Member(I,Create)),
--| Member(I,Insert(L,J,N)) = ((I = J) or Member(I,L)),
```

Step VIII Define each generator $g \in Gen(T) - GenBas(T)$ (which has not been defined yet) in terms of the defined observers or the defined generators.

Again, there are two options. Either, expressions of the form $o(g(X))$ should be defined in terms of other expressions, for all $o \in$ basic observers. Or, $g(g2(X)) = g2(g(X))$ should be an axiom, for all $g2 \in$ basic generators. The method followed is the same as that in step **VI** in both cases.

```
--| Length(Delete(L,N)) = Length(L) - 1,
--| Get(Delete(L,N),M) =
--|    if N > M then Get(L,M) else Get(L,M+1) end if;
-- Or.
--| Delete(Insert(L,I,N),M) =
--|    if N + 1 = M then L
--|    elsif M <= N then Insert(Delete(L,M),I,N-1)
--|    else Insert(Delete(L,M-1),I,N) end if;
```

3.1 Subset of Packages to which the Methodology Applies

This methodology can be applied to packages with procedures by converting each procedure to a number of functions. The first function returns a record with all the out parameters of the procedure and the other functions select each out parameter from this record. Packages with internal state are handled by converting each function to a procedure, adding an "in out" state parameter and converting the procedure back to functions (by the above method). Objects are coverted into pairs of procedures which read and write a component of the state. The methodology however does not handle concurrency (tasks) in the package specification.

There can be other specifications which are more succinct than the specifications resulting from this methodology. Also, the user may not be able to define the meaning of $o(g(X))$ for all cases. For example, consider Example 1. The square root function requires infinite equations (one for each number) for its definition according to our methodology. However, the methodology we have presented applies to enough packages to make it interesting.

4 Correctness of Formal Specifications

The next step is to gain a high degree of confidence in the correctness of the specification. Here by the term *correctness* it is meant that the specification reflects the behavior intended by the specifier. This confidence should be gained as early as possible, preferably before an actual implementation has been built.

A specifier needs confidence in two areas of specification:

- His internal, informal view of the specification of a problem should be a satisfactory expression of the problem itself, and
- The formal specification should implement this internal view correctly.

Deductive tools may be used to increase the specifier's level of confidence in both areas. One such tool, the Anna Package Specification Analyzer[10], has been built for just such a purpose. The Specification Analyzer builds a model of the specification as a logic state, i.e. a set of logical expressions concerning the relationships between subprograms defined by the package. It contains a deductive theorem prover to derive logical consequences of the specification. The user then asks questions with respect to this state, such as the value of an expression as implied by package axioms. The state serves as a prototype implementation of the package, allowing the user to test ideas and if necessary to alter the specification to reflect either changes in the user's internal view of the problem, or errors in the specification.

For example, recall the five methods of defining the observer function expressions in Section 3. If the observer functions of the List package have been defined by these methods, using Specification Analysis it is possible to test that the value of the observers are what were intended. A user may query the value of a specific observer; and if the annotations define the function in terms of simple discrete-valued functions, and are complete enough to define a unique result, that result may be calculated. Using the List example, assume variables I1 and I2 of type Item have been declared (Ada variables may be declared interactively during Specification Analysis, to store temporary values). On passing the following expression to the Specification Analyzer,

    Length(Insert(I2,1,Insert(I1,0,Create)))

if the Length and Insert functions are defined following methods **1** and **2**, the Analyzer would deduce that the value of this expression is 2. If the evaluation of an observer terminates abnormally, as in method **3**, this will be reported to the user. If no value for the expression is deducible, the user must decide whether this incompleteness was intentional, as in method **5**, or whether the specification is incorrect.

Where equivalence classes of functions are defined, as in method **4** or where functions are not defined in terms of other discrete valued functions, the user queries not specific function calls but various equivalences, to see if they hold. Using the Integers package example, a user may query expressions such as:

$$\text{Succ(Succ(Zero))} \;=\; \text{Pred(Succ(Succ(Succ(Zero))))}$$

and the result will be one of the boolean values True or False, or that neither result is provable.

Specification Analysis does not decide which function expressions to evaluate — developing test cases is still up to the user. But it does allow the user to check all expressions necessary to increase confidence in the correctness of the specification.

5 A Methodology for Implementing Package Specifications

Given a package which has been formally specified by following the above methodology, the methodology outlined below results in a prototype Ada implementation of the package.

Step I Choose the observers or the generators to be the basis of the data structure which implements a type T in $NewT$. Choose the observer basis if most functions are specified in terms of it, else choose the generator basis. Go to step I.1 in the former case and follow the ".1" steps and to step I.2 in the latter case and follow the ".2" steps.

Step I.1 If there are N basic generators $g_1, \ldots, g_N$ for type T, choose a variant record with N variants. The ith variant has m components $t_{i_1}, \ldots, t_{i_m}$ corresponding to the parameter types of the ith generator $g_i : t_{i_1}, \ldots, t_{i_m} \rightarrow T$. In the general case where a generator for T might have a parameter of type T, access (pointer) types should be used. For example, for lists:

```
type List_Types is (Inserted_List, Created_List);
type List_Record(Kind : List_Types);
type List is access List_Record;
type List_Record(Kind : List_Types) is record
      case Kind is
            when Inserted_List =>
                  Elem : Item; Next : List;
                  Number : Natural;
            when Created_List => null;
      end case;
end record;
```

In the following we assume that the functions which are not basic observers or basic generators are specified in terms of basic generators. Skeleton implementations are provided for the case where these functions may have been defined in terms of the basic observers.

Step II.1 Implement generator basis functions. Implementation of $g_i : t_{i_1}, \ldots, t_{i_m} \rightarrow T$ returns a new variant record value which has the ith variant and which is assigned the values of parameters t_i of g. For example:

```
function Create return List is
begin
```

```
      return new List_Record(Created_List);
   end Create;
   function Insert(L : List; I : Item;
                   After : Natural) return List is
   begin
      return new List_Record'(Inserted_List,I,L,After);
   end Insert;
```

Step III.1 Implement the observer basis functions. This step is dependent on the axioms which relate the observer and generator basis. The implementation mimics the axioms. For each option in step **VI** of the specification methodology, we have an option here:

1. If $o(g(X)) = o(X)$ is the axiom in the specification the implementation of $o(g(X))$ is a recursive call $o(X)$. For example:

```
function Length(L : List) return Natural is
begin
   case L.Kind is
      when Created_List => return 0;
      when Inserted_List =>
         return 1 + Length(L.Next);
   end case;
end Length;
function Get(L : List; N : Positive) return Item is
begin
   case L.Kind is
      when Created_List =>
         raise Out_Of_Range;
      when Inserted_List =>
         if L.Number + 1 = N then
            return L.Elem;
         elsif N <= L.Number then
            return Get(L.Next,N);
         else return Get(L.Next,N-1);
         end if;
   end case;
end Get;
```

The exception is raised because of the existence of an exception propagation annotation in the specification.

2. Follow option 1 if a subprogram result annotation is used.

3. If $o(g(X))$ is specified to terminate abnormally with an exception, the implementation raises an exception. If $o(g(X))$ is specified to just terminate abnormally, the implementation is free to abnormally terminate in any way. See option 1 for an example.

4. Equations between basic generators do not substantially affect the methodology for implementation. All they do is to suggest improvements in the data structure chosen, since many values of the data structure are now the same.

5. If $o(g(X))$ is undefined, the user is free to implement o in any way (s)he desires, including not implement it.

Step IV.1 Implement other observers based on corresponding axioms in step **VII** of the specification methodology. If the specification of o is through an equation $o(X) = o2(X)$ then the implementation involves a call to $o2(X)$. For example:

```
function Member(I : Item; L : List) return Boolean is
begin
    for N in 1 .. Length(L) loop
        if Get(L,N) = I then return True; end if;
    end loop;
end Member;
```

If the specification defines $o(g(X))$ for all $g \in GenBas(T)$ then the implementation follows step III.1 above:

```
function Member(I : Item; L : List) return Boolean is
begin
    case L.Kind is
        when Created_List => return False;
        when Inserted_List =>
            return (L.Elem = I) or Member(I,L.Next);
    end case;
end Member;
```

Step V.1 Implement other generators. Look at the axiom that defines the generator g. If the axiom is $o(g(X)) = o2(X)$, the implementation of g returns a record the value of observers o on which is $o2(X)$. As pointed out at the end of step I.1, we do not fully treat the case where observers are used to define g.

If the axiom defines $g(g2(X))$, the implementation of $g(Y)$ involves a call to $g2(g(X))$. This is simple since any actual parameters Y in a call $g(Y)$, are already in the form $g2(X)$. For example:

```
function Delete(L : List; N : Positive) return List is
begin
    if L.Kind = Created_List then
        raise Out_Of_Range;
    elsif L.Number + 1 = N then
        return L.Next;
    elsif N <= L.Number then
        return new List_Record'
        (Inserted_List,L.Elem,Delete(L.Next,N),L.Number-1);
    else
        return new List_Record'
        (Inserted_List,L.Elem,Delete(L.Next,N-1),L.Number);
    end if;
end Delete;
```

Step I.2 For each $o : t_1, \ldots, T, \ldots, t_n \rightarrow X$ in the basic observers, where t_is are types of parameters and X is the return type, define a multidimensional array which has elements of type X and is indexed by the t_is. There could be two problems with this—any t_i could be very large (like Integer) or any t_i could be a non-discrete type (like Float). The way to resolve both these problems is to instead choose a linked structure. An array indexed by types $t*$ is replaced by a linked list of records with components of types $t*$ and X. Each element of this list corresponds to an element of the array. In the following we assume the existence of operations on linked lists corresponding to the operations "$A(I)$" and "$A(I) := X$" on arrays. It is assumed in later discussion that these operations are available. For the list example we define two arrays corresponding to the two basic observers:

```
type Get_Record;
type Get_Array is access Get_Record;
type Get_Record is record
    Elem : Item; Index : Positive; Next : Get_Array;
end record;
type List is record
    Len : Natural; Get_Obs : Get_Array;
end record;
```

Len is a constant "array" corresponding to Length. Get_Obs is a one dimensional "array" corresponding to Get.

In the following we assume that functions which are not basic observers or basic generators, are specified in terms of the basic observers. We do not treat fully the cases where basic generators are used to specify these functions.

Step II.2 Implement observer basis functions. Observer o is implemented by the selection operation "$A(I)$" on the array corresponding to o. For example:

```
function Length(L : List) return Natural is
begin
    return L.Len;
end Length;
function Get(L : List; N : Positive) return Item is
begin
    -- Get the Elem field of the node of the Get_Obs linked list which
    -- has Index = N.
end Get;
```

Step III.2 Implement basic generators. The implementation mimics the axioms relating basic observers and generators defined in step VI of the specification methodology.

1. If $o(g(X)) = o(X)$ is the defining axiom, the implementation of g(X) returns a record with values corresponding to $o(X)$. For example:

   ```
   function Create return List is
   begin
       return (0,null);
   end Create;
   function Insert(L : List; I : Item;
                   After : Natural) return List is
   begin
       -- Return a new list which is the same as L except that 1. Length
       -- is incremented by 1, 2. Indices of all elements of the old list
       -- where Index > After have been incremented by 1, 3. A new
       -- element I with Index = After+1 has been added.
   end Insert;
   ```

2. If $o(g(X))$ terminates abnormally, terminate abnormally.

3. Equations between generators do not affect implementation.

4. Partial specification allows any implementation.

Step IV.2 Implement other generators. If the defining axiom for the generator defined $o(g(X))$ then steps similar to step III.2 should be taken. For lists:

```
function Delete(L : List; N : Positive) return List is
begin
    -- Return a new list which is the same as L except that 1. Length
    -- is decremented by 1, 2. Indices of all elements of the old list
```

```
      -- where Index > N have been decremented by 1, 3. Element
      -- with Index = N has been removed.
   end Delete;
```

If $g(g2(X)) = g2(g(X))$ is defined, then the implementation of $g(Y)$ involves rewriting Y as a generator expression $g2(X)$ and then calling $g2(g(X))$. That is, choosing an X such that $Y = g2(X)$. For the list example this involves setting a general list L equal to some list $Insert(L1, I, N)$ and then calling $Insert(Delete(\ldots))$. As pointed out at the end of step II.1 above, we do not fully treat the case where o is defined by basic generators.

Step V.2 Implement other observers. If the defining axiom is of the form $o(X) = o2(X)$, the implementation of $o(X)$ is a call to $o2(X)$. For example:

```
   function Member(I : Item; L : List) return Boolean is
   begin
      -- Check if Elem I exists in list.
   end Member;
```

If the defining axiom is $o(g(X)) = o2(X)$, the implementation of $o(Y)$ involves rewriting Y as $g(X)$ and making a call to $o2(X)$ similar to step IV.2. We do not treat this case fully.

5.1 Limitations of the Methodology

The subset of specifications to which the methodology applies may be too restrictive in some cases. The same problems that disallow specification of the square root function in our methodology, also disallow implementation according to our methodology.

A major shortcoming is that the implementation obtained is not the most efficient. Various optimizations to the data structures used in the methodology give rise to stacks, sets, queues and other structures. See [13] for an overview of data structure selection for implementations.

The prototype implementation allows a user to get insight into the properties of the final implementation and also to test out properties of a program before implementing the program in its final form.

6 Correctness of Implementations

The formal specification of a package constrains the behavior of any implementation written for it. Correctness with respect to an implementation means that executions of the implementation satisfy the package constraints. Given an implementation, different levels of verifying its correctness are possible. Deductive tools may prove from lower-level Anna specifications, or from the Ada code itself, that an implementation obeys the package constraints[8].

Another option is run-time verification[14]. Here, a tool inserts run-time Ada checks into the implementation which test the constraints. If a check fails, the specific annotation violated and location in the body where it was violated may be reported for debugging. Hence permanent low-cost checking can ensure that every execution of a package body obeys its specification.

7 Conclusion

We have presented a methodology for specification and implementation of Ada packages. The motivation for obtaining a specification and a prototype implementation have been pointed out. The methodology is applicable to most Ada packages. It is our experience that apart from certain cases like the square root function, most packages can be specified by our methodology.

In these cases, the use of Anna *virtual functions* may be made to introduce intermediate functions which bridge the gap between the two specifications. We are involved in ongoing research which investigates this line of thought.

Future research will involve extending the methodology to include more kinds of specifications and implementations and to develop rules for optimization of implementations. Future research will also involve development of tools which help in applying the methodology to actual packages and tools which automate parts of specification and implementation of packages. Proofs obligations for each step of implementation in the spirit of [5] will also be developed.

Acknowledgements

We would like to thank Professor David Luckham at Stanford for his guidance and his support for writing this paper.

This research was supported by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211.

References

[1] H. K. Berg, W. E. Boebert, W. R. Franta, and T. G. Moher. *Formal Methods of Program Verification and Specification*. Prentice-Hall, 1982.

[2] R. Boyer and J Strother Moore. Program Verification. *Journal of Automated Reasoning*, 1(1):17–22, 1985.

[3] J. V. Guttag and J. J. Horning. The Algebraic Specification of Abstract Data Types. *Acta Informatica*, 10:27–52, 1978.

[4] J. C. Huang. Error Detection Through Program Testing. In *Current Trends in Programming Methodology, Volume II—Program Validation*. Prentice-Hall, 1977.

[5] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 1986.

[6] Barbara Liskov and Stephen Zilles. An Introduction to Formal Specifications of Data Abstractions. In *Current Trends in Programming Methodology, Volume I—Software Specification and Design*. Prentice-Hall, 1977.

[7] Ralph L. London. Perspectives on Program Verification. In *Current Trends in Programming Methodology, Volume II—Program Validation*. Prentice-Hall, 1977.

[8] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford Pascal Verifier user manual. Technical Report 79-731, Department of Computer Science, Stanford University, March 1979.

[9] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. *Anna—A Language for Annotating Ada Programs.* Springer-Verlag—Lecture Notes in Computer Science No. 260, July 1987.

[10] Walter R. Mann. Anna package specification analyzer user's guide. Unpublished technical report.

[11] D. L. Parnas. A Technique for Software Module Specification with Examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[12] Charles Rich and Richard C. Waters, editors. *Artificial Intelligence and Software Engineering.* Morgan Kaufman Publishers, 1986.

[13] Lawrence A. Rowe and Fred M. Tonge. Automating the Selection of Implementation Structures. *IEEE Transactions on Software Engineering*, SE-4(6), 1978.

[14] Sriram Sankar. *Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs.* PhD thesis, Stanford University, 1989.

[15] United States Department of Defense. *Reference Manual for the ADA Programming Language.* ANSI/MIL-STD-1815A-1983.

A Formal Specification and Implementation of the List Package

One of the possible formal specifications of the list package which results from the methodology for specifications is presented here. The operations Member and Delete are specified in terms of first the generator basis and then the observer basis. The data structures are provided for two cases—implementation choosing the observer or the generator basis.

```
generic
    type Item is private;
package List_Package is
    type List is private;
    Out_Of_Range : exception;
    function Length(L : List) return Natural;
    function Get(L : List; N : Positive) return Item;
    --| where (Length(in L) < N) =>
    --|         raise Out_Of_Range;
    function Create return List;
    function Insert(L : List; I : Item;
                    After : Positive) return List;
```

```
    --| where (Length(in L) < After) => raise Out_Of_Range;
    function Delete(L : List; N : Positive) return List;
    --| where (Length(in L) < N) => raise Out_Of_Range;
    function Member(I : Item; L : List) return Boolean;
    --| axiom for all I, J : Item; L : List;
    --|                N, M : Natural =>
    --|      Length(Create) = 0,
    --|      Length(Insert(L,I,N)) = 1 + Length(L),
    --|      Get(Insert(L,I,N),M) =
    --|           if N + 1 = M then I
    --|           elsif M <= N then Get(L,M)
    --|           else Get(L,M-1) end if,
    --|      not(Member(I,Create)),
    --|      Member(I,Insert(L,J,N)) =
    --|           ((I = J) or Member(I,L)),
    --|      Delete(Insert(L,I,N),M) =
    --|           if N + 1 = M then L
    --|           elsif M <= N then
    --|                Insert(Delete(L,M),I,N-1)
    --|           else Insert(Delete(L,M-1),I,N) end if;
    -- Or.
    --|      Member(I,L) = (exist I : 1 .. Length(L) =>
    --|           Get(L,N) = I),
    --|      Length(Delete(L,N)) = Length(L) - 1,
    --|      Get(Delete(L,N),M) =
    --|           if N > M then Get(L,M)
    --|           else Get(L,M+1) end if;
private
    type List_Types is (Inserted_List, Created_List);
    type List_Record(Kind : List_Types);
    type List is access List_Record;
    -- Or.
    type Get_Record;
    type Get_Array is access Get_Record;
    type Get_Record is record
         Elem : Item;
         Index : Positive;
         Next : List;
    end record;
    type List is record
         Len : Natural;
         Get_Obs : Get_Array;
    end record;
```

```
end List_Package;
```

Implementation using the generator basis for choosing the data structure to implement lists:

```
package body List_Package is
    type List_Record(Kind : List_Types) is record
        case Kind is
            when Inserted_List =>
                Elem : Item; Next : List;
                Number : Natural;
            when Created_List => null;
        end case;
    end record;
    function Create return List is
    begin
        return new List_Record(Created_List);
    end Create;
    function Insert(L : List; I : Item;
                After : Positive) return List is
    begin
        return new List_Record'(Inserted_List,I,L,After);
    end Insert;
    function Length(L : List) return Natural is
    begin
        case L.Kind is
            when Created_List => return 0;
            when Inserted_List =>
                return 1 + Length(L.Next);
        end case;
    end Length;
    function Get(L : List; N : Positive)
                                        return Item is
    begin
        case L.Kind is
            when Created_List =>
                raise Out_Of_Range;
            when Inserted_List =>
                if L.Number + 1 = N then
                    return L.Elem;
                elsif N <= L.Number then
                    return Get(L.Next,N);
                else
                    return Get(L.Next,N-1);
                end if;
        end case;
```

```
    end Get;
    function Member(I : Item; L : List) return Boolean is
    begin
        case L.Kind is
            when Created_List => return False;
            when Inserted_List =>
                return (L.Elem = I) or Member(I,L.Next);
        end case;
    end Member;
    function Delete(L : List; N : Positive)
                                        return List is
    begin
        if L.Kind = Created_List then
            raise Out_Of_Range;
        elsif L.Number + 1 = N then
            return L.Next;
        elsif N <= L.Number then
            return new List_Record'
                (Inserted_List,L.Elem,
                 Delete(L.Next,N),L.Number-1);
        else
            return new List_Record'
                (Inserted_List,L.Elem,
                 Delete(L.Next,N-1),L.Number);
        end if;
    end Delete;
end List_Package;
```

Implementation using the observer basis for choosing the data structure to implement lists:

```
package body List_Package is
    function Length(L : List) return Natural is
    begin
        return L.Len;
    end Length;
    function Get(L : List; N : Positive)
                                        return Item is
            Tmp : Get_Array := L.Get_Obs;
    begin
        while Tmp /= null loop
            if Tmp.Index = N then
                return Tmp.Elem;
            else
                Tmp := Tmp.Next;
```

```
          end if;
      end loop;
      raise Out_Of_Range;
end Get;
function Create return List is
begin
     return (0,null);
end Create;
function Insert(L : List; I : Item;
                After : Positive) return List is
     L1 : List := L;
     Tmp : Get_Array := L1.Get_Obs;
begin
     L1.Len := L1.Len + 1;
     while Tmp /= null loop
          if Tmp.Index > After then
               Tmp.Index := Tmp.Index + 1;
          end if;
          Tmp := Tmp.Next;
     end loop;
     L1.Get_Obs := new Get_Record'(I,After+1,L1.Get_Obs);
     return L1;
end Insert;
function Member(I : Item; L : List)
                              return Boolean is
begin
     for N in 1 .. Length(L) loop
          if Get(L,N) = I then
               return True;
          end if;
     end loop;
end Member;
function Delete(L : List; N : Positive)
                                     return List is
     L1 : List := L;
     Tmp, Tmp2 : Get_Array;
begin
     L1.Len := L1.Len - 1;
     if L1.Get_Obs.Index = N then
          L1.Get_Obs := L1.Get_Obs.Next;
     end if;
     Tmp := L1.Get_Obs;
     while Tmp /= null loop
```

```
            if Tmp.Index > N then
                Tmp.Index := Tmp.Index - 1;
            elsif Tmp.Index = N then
                Tmp2.Next := Tmp2.Next.Next;
            end if;
            Tmp2 := Tmp;
            Tmp := Tmp.Next;
        end loop;
        return L1;
    end Delete;
end List_Package;
```